Toward Reusable Graphics Components in Ada

THESIS

Sam-kyu Lim
Captain, ROKAF

AFIT/GCS/ENG/93M-03

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/93M-03

Toward Reusable Graphics Components in Ada

THESIS

Sam-kyu Lim
Captain, ROKAF

AFIT/GCS/ENG/93M-03

AFIT/GCS/ENG/93M-03

Toward Reusable Graphics Components in Ada

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Systems

Sam-kyu Lim, B.S.
Captain, ROKAF

March 1993

Preface

The purpose of this thesis was to investigate and to demonstrate Ada's applicability as an implementation language for a reusable graphical software component.

This thesis was accomplished through many people's encouragement and support. I would like to express my deep appreciation to those people who helped me during the last 21 months.

First of all, I would like to thank my thesis advisor, Lieutenant Colonel Patricia K. Lawlis for her help and guidance during this thesis. This would not have been possible without her help. I also want to thank my thesis committee members, Major David R. Luginbuhl and Captain Dawn A. Guido. Their continuous guidance made it possible for me to complete my thesis. Additional thanks goes to Dr. Hyong Lee. He helped my thesis work by checking grammar and solving miscellaneous problems.

My greatest thanks goes to my wife Woojung and son Jonghyun. Thanks for sharing all difficulties, giving their devoted support, and for enduring a long lonely time.

Sam-kyu Lim

Table of Contents

List of Figures

List of Tables

AFIT/GCS/ENG/93M-03

Abstract

This thesis demonstrates and illustrates a way of developing reusable graphics software components in Ada associated with a C++/C library. The work was carried out using object-oriented software development techniques that were used to analyze, design and implement a partial flight simulator. The objective of this thesis was to present a way of building reusable software components with Ada in a graphics application environment.

An object-oriented approach was taken in the development of a set of reusable graphics software components for a flight simulator domain. A selection of a set of reusable software components came from domain analysis. These components were analyzed in detail, then redesigned to demonstrate and illustrate the thesis objective. Examples from design and implementation demonstrate how Ada 83 was applied in building reusable graphics software components associated with C++ routines, the limitations of Ada 83, and how Ada9X addresses these limitations.

Toward
Reusable Graphics Components in Ada

I. Introduction

1.1 Background

The problems with software today are typically associated with what some people call the "software crisis". "Over the past ten years, the software crisis has cost the Department of Defense alone tens of billions of dollars" (42:2). As software development and maintenance costs have continued to rise at ever increasing rates, the use of standards as a solution to the problem has been suggested. Standardization is far more important in the drive toward reuse than special mechanisms particular languages may offer (14). Ada has been established as the single, standardized programming language within the U.S. Department of Defense (DoD). The aim was to replace the vast number of languages and ad hoc techniques previously in use with a standard programming paradigm to promote maintainability, reusability, and portability. Program portability and reuse of software components are a major concern of the DoD as well as the software industry since both use a wide variety of hardware platforms. However, a problem introduced by the newness of Ada is a lack of potential experience in several key areas. For example, the effective use of Ada in solving graphics problems with DoD requirements.

A problem of this type is evident in the flight simulator software that was partially developed by several Air Force Institute of Technology (AFIT) students (15, 26, 37). The program was written in C++ rather than Ada, and it exhibits many of the engineering difficulties that prompted the development of Ada. This thesis will investigate the use of Ada to replace C++/C in a graphics application.

1.2 Problem Statement

A problem with the use of Ada is a lack of potential experience in several key areas such as the effective use of Ada in solving graphics problems with DoD requirements.

1.3 Hypothesis

The hypothesis of this thesis was that the advanced features of the Ada programming language could be successfully interfaced with the advanced graphics libraries of a Silicon Graphics System and existing C++ code.

1.4 Research Objective

The overall goal of this research was to investigate and to demonstrate Ada's applicability as an implementation language for a reusable graphical software component using an object-oriented approach. During the course of the investigation, two of the following Ada features were examined:

- What problems exist with Ada 83 in building reusable graphics software components using an object-oriented approach, and how these problems are addressed in Ada 9X,
- How to connect Ada programs to existing C++ code.

1.5 Scope

The design and implementation of parts of a flight simulator will serve as the foundation for the investigation of Ada's applicability as an implementation language for graphics programs.

1.6 Assumptions

1.6.1 Hardware. A flight simulator written in C++ was executed on a Silicon Graphics IRIS (SGI), so access to an SGI was necessary.

1.6.2 Software. We had access to the following sources of software:

- Ada: The Verdix Ada Development System (VADS) was used. VADS provides users with a complete software environment for developing Ada language applications. VADS provides users with useful libraries for interfacing to the Silicon Graphics Graphics Library, Font Manager Library, operating system calls, etc.
- The C++ programming language: The AT/T C++ translator release 2.0 was used.
- Unix Standard Library: We were working within the Unix operating system. We therefore were able to use Unix standard system calls and "C" software libraries.
- Previous Thesis Efforts: The work done by Captain Simpson (37) for controlling the input devices was reused in this thesis.

1.7 Approach/Methodology

An incremental approach was taken in the development of this thesis project. First, the domain of the flight simulator was analyzed and a set of components was selected. Then the existing C++/C code and the object-oriented model were analyzed for design of reusable software components. As a part of this, Ada bindings to C++ and C were analyzed. Then alternatives for obtaining well-engineered reusable software components were examined. Finally, the alternatives were analyzed, and the most effective one was chosen and implemented.

1.8 Materials and Equipments

The following materials and equipment were used in this research.

- Target Machine : Silicon Graphics IRIS 4D Series
- Joystick : Microstick Joystick from CH Products

1.9 Summary

As software development and maintenance costs have continued to rise at ever increasing rates, the use of standards as a solution to the problem has been suggested. Ada has been established as the single, standardized programming language within the U.S. DoD to replace the vast number of languages and ad hoc techniques previously in use to promote reusability, maintainability, and portability. This thesis examines previous thesis work on flight simulator software implemented in C++, and then builds on this work using Ada.

The objective of this thesis is to demonstrate and illustrate the feasibility of building well-engineered reusable graphics software components in Ada.

1.10 Thesis Overview

This document contains five chapters. Chapter 2 is a literature review of object-oriented technology, software reuse, and the contrasts of Ada and C++/C. Chapter 3 outlines the process used to examine the problem as well as the design of reusable software components. Chapter 4 describes detailed design and implementation strategies in building a well-engineered reusable set of components in conjunction with C++ routines in a class library. Chapter 5 includes a summary and conclusions.

II. Literature Review

2.1 Introduction

The purpose of this chapter is to aid in the understanding of the discussion contained in the following chapters. This chapter begins with a description of the need for object-oriented techniques, followed by concepts of the object-oriented paradigm with the basic analysis and design concepts. This description is followed by a description of software reuse, especially focusing on reuse of code components. The features of Ada and C++/C are then considered for similarities and contrasts. This is followed by a description of interfacing Ada with C++/C and a description of the MICROSTICK device used in this study.

2.2 The Object-Oriented Paradigm

2.2.1 The Need for Object-Oriented Techniques. Among software engineers the software crisis is a well known fact (8, 29). "The essence of the software crisis is simply that it is much more difficult to build software than our intuition tells us it should be" (8:7). In general, the software crisis is characterized by many problems, while managers' responses for software development concentrate on the "bottom-line" issues such as: 1) inaccurate schedule and cost estimates; 2) the lack of "productivity" from software engineers; and 3) the lack of software quality (29). The major cause of the software crisis is the complexity of software and software itself (7). The previous situation highlighted the need for developing and maintaining large complex software systems in a competitive and dynamic environment, and it has driven interest in new approaches to software design and development. The goals (*modifiability, efficiency, reliability, understandability*) and principles (*modularity, abstraction, localization, information hiding, uniformity, completeness, confirmability*) of software engineering as stated by Goodenough, Ross, and Irvine (16) and Booch (8), provide the foundation upon which software will be designed in the future. "Object-oriented design, object-oriented programming, and object-based programming are methods that support the goals and principles of software engineering" (31:76).

2.2.2 Object-Oriented (OO) Paradigm.

Although many discussions have taken place, little consensus has been reached as to exactly what is meant by "object orientation", even among leaders like Booch (7), Meyer (24), and Stroustrup (38). Henderson and Edwards (17:146) define the object-oriented paradigm as follows: "The object-oriented (OO) paradigm, at its simplest, takes the same components of a software system: data and procedures, but de-emphasizes the procedures, stressing instead the encapsulation of data and procedural features together, exemplified by the clear and concise specification of the module interface". The paradigm sprang from language, has matured into design, and has recently moved into analysis. Several general concepts that are strongly related with the OO paradigm will be discussed in this chapter: objects, classes, abstraction, inheritance, encapsulation, polymorphism, and dynamic binding. Although these concepts are basic to the object-oriented paradigm, the various object-oriented communities often associate different specifics with each concept. This section focuses on those concepts.

2.2.2.1 Objects.

Booch (7) defines an object as follows:

> An object is a tangible or visible entity that exhibits some well-defined behavior. It has state, behavior, and a unique identity; the state of an object encompasses all of the properties of the object plus the current values of each of these properties, the behavior is how an object acts and reacts, in terms of its state changes and message passing, and identity is that property of an object which distinguishes it from all other objects.

Objects, well-defined and limited, may contain both data structures and actions to be performed on the structures. The basic principles behind objects are: 1) abstraction, 2) encapsulation, 3) modularity, 4) hierarchy, and 5) polymorphism (27).

1) Abstraction is the process of defining an object in just the right amount of detail for the situation. The goal is to use just enough detail to differentiate a new object from other objects at the same level. The easiest way to do this is to portray the operations the object can perform before worrying about operations within the object.

2) Encapsulation involves hiding information about an object from the user. It hides the details of the implementation because the user doesn't care how operations are implemented, only how to use them.

3) Modularity follows the principle of divide and conquer: Divide the problem into pieces and keep each piece of the solution in independent modules. The object is perfect for modularity because it holds the data structures and operations for each part of the solution. The module is also loosely coupled because it is not tied closely to other parts of the program.

4) Hierarchy means that objects are abstracted in terms of the problem to be solved, then described in more specific terms, with the objects adding characteristics or inheriting others from existing objects.

5) Polymorphism refers to the ability of an object to assume multiple roles and shapes. For instance, "overloading" is a fancy term for the simple practice of using an operator for more than one type of data.

2.2.2.2 Classes. A class is a set of objects that share a common structure and common behavior. Korson (21:42) defined a class as follows:

> Ideally, a class is an implementation of an abstract data type. This means that the implementation details of the class are private to the class. The public interface of such a class is composed of two kinds of class methods. The first kind consists of functions that return meaningful abstractions about an instance's state. The other kinds of methods are transformation procedures used to move an instance from one valid state to another.

By grouping objects into classes, a problem can be abstracted. Class is the language construct most commonly used to define abstract data types in object-oriented programming languages.

2.2.2.3 Abstraction. Abstraction is the fundamental concept of object orientation. "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer" (7:39). Abstraction is one of the fundamental ways that we humans cope with complexity. Abstraction encourages programmers and users to think about complex applications in abstract terms. "The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant" (32:16). There are two methods of abstraction: abstraction by specification and abstraction by parameterization (21).

Abstraction by specification abstracts the specification of an entity from its implementation. This type of abstraction is supported by virtually every object-oriented language. The public interface of a class constitutes the specification of that class. The interface specifies the legitimate operators of the data contained in instances of the class.

Abstraction by parameterization abstracts the type of data to be manipulated from the specification of how it is to be manipulated. This type of abstraction is supported by most object-oriented languages at the operator level, but by only a few languages at the class level.

2.2.2.4 Inheritance. Inheritance is a relation between classes. "Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship" (32:3). It is not provided by conventional languages. "Inheritance is a way of defining some useful construct in a central place and then automatically broadcasting that construct to all the places where it could help. New functionality is no more developed by coding each line from scratch, but by inheriting some useful class and describing only how the new one differs" (13:12). Inheritance not only supports reuse across systems, but it directly facilitates extensibility within a given system.

2.2.2.5 Encapsulation. Encapsulation is the way of building of methods and data together within an object so that access to data is permitted only through the object's own methods (2:35). "Abstraction and encapsulation are complementary concepts: abstraction focuses upon the outside view of an object and *encapsulation* - also known as information hiding - prevents clients from seeing its inside view, where the behavior of the abstraction is implemented" (7:45).

2.2.2.6 Polymorphism. As Cardell (9) points out there are many kinds of polymorphism, but in general, polymorphism means the ability to take more than one form. The same operation may apply to many different classes, that is, the same operation takes on different forms in different classes. Because of this ability to refer to more than one class of object, a polymorphic reference has both a dynamic and a static type associated with it (21).

The dynamic type of a polymorphic reference may change from instant to instant during the program execution. In strongly typed object-oriented environments, the run-time system keeps all polymorphic references automatically tagged with their dynamic type.

The static type is determined from the declaration of the entity in the program text. It is known at compile time and determines the set of valid types that the object can accept at run-time.

2.2.2.7 Dynamic Binding. The binding discussed in this chapter is the binding of a procedure call to the code to be executed in response to the call. Dynamic binding means that binding is done later than compile-time, generally while the program is running. "Dynamic binding is needed in loosely coupled collections where the customer's code cannot predict the type of data to be operated on until the code is being run" (13:14). Dynamic binding is intrinsic to the very essence of a loosely coupled collection. "In the object-oriented world, dynamic binding is associated with polymorphism and inheritance in that a procedure call associated with a polymorphic reference may depend on the dynamic type of that reference" (21:46).

2.2.2.8 Terminology. This discussion introduces key concepts and definitions from the OO domain. Like any emerging technologies, OO has many proponents with differing opinions. Since object-oriented methodologies are in their early stage, like the various object-oriented programming languages, terminology for the object-oriented mechanisms differs among methodologies (2:189).

2.2.3 Object-Oriented System Life Cycle. "At the most general level, three phases to the life cycle are generally agreed upon: 1) analysis, 2) design and 3) construction/implementation" (17:143). Like a general software development life cycle, an object-oriented development life cycle also has analysis, design, and implementation phases. Problems with traditional development using the classical life cycle include no iteration, no emphasis on reuse, and no unifying model to integrate the phases (21). In contrast to the common structured systems analysis and design based largely on top-down functional decomposition, object-oriented analysis and design has many attributes of both top-down and, perhaps predominantly, bottom-up design. Since one of the aims of an OO implementation is the development of generic classes for storage in libraries, an approach which considers both top-down analysis and bottom-up design simultaneously is likely to lead to the most robust software systems (17:146). The object-oriented analysis and the object-oriented design phases work more closely together because of the commonality of the object model. In one phase, the analyst identifies problem domain objects while in the next phase, the designer specifies additional objects necessary for a specific computer-based solution. The design process is repeated for these implementation-level objects.

2.2.3.1 Basic Analysis Concepts. "Object-Oriented Analysis (OOA) is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain" (7:37). It is concerned with constructing a precise, concise, understandable, and correct model of the real world. "The analysis model serves several purposes. It clarifies the requirements, it provides a basis for agreement between the software requestor and the software developer, and it becomes the framework for later design and implementation" (32:148). There are several OOA approaches (7, 10, 17, 32, 35) each with their own techniques. These methods can be summarized by the following activities, which may overlap.

- Identify the classes/objects of problem space
- Identify the relationship between classes

- Identify the attributes and methods of each class/object
- Specify the inter-object communication.

2.2.3.2 Basic Design Concepts. This section describes the fundamentals of the object-oriented design phase of the object-oriented software life cycle. The different point of view between the procedural design paradigm (top-down functional decomposition) and object-oriented design is that the procedural paradigm takes a task-oriented point of view, while the object-oriented design paradigm takes a modeling point of view (21). Booch defines object-oriented design (OOD) as follows. "Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design" (7:37). The application design process begins at a top level and proceeds through class identification to a low level and then moves upward as low-level classes are designed based on lower-level definitions. The object-oriented paradigm provides support for good design: 1) modularity, 2) information hiding, 3) weak coupling, 4) strong cohesion, 5) abstraction, 6) extensibility, and 7) integration (21). There are many sources of advice on what makes a good design (10, 32).

2.2.3.3 Notation. To do analysis and design, we need a way to picture the things we want to build, a notation for modeling the structure of object-oriented software. "Any graphical representation of the object-oriented version of the overall software development life cycle must take into account the high degree of overlap and implicit iteration" (17:151) Various notations have been introduced by various authors (7, 10, 24, 32).

2.2.4 Benefits and Drawbacks of the Object-Oriented Approach. Like a classical approach, the object-oriented approach also has its benefits and drawbacks.

2.2.4.1 Benefits. The object-oriented paradigm offers the following benefits: 1) a way to manage complex software, 2) a "seamless" way to perform analysis, design and implementation, 3) reusability, 4) maintainability and 5) extensibility (37:2-28). Booch noted the following benefits by applying object-oriented design (7).

- Exploits the expressive power of all object-based and object-oriented programming languages
- Encourages the reuse of software components
- Leads to systems that are more resilient to change
- Reduces development risk
- Appeals to the working of human cognition

2.2.4.2 Drawbacks. Although the OO approach has valuable benefits, it also has some drawbacks that must be considered. There two acknowledged drawbacks to using the object-oriented approach: 1) performance considerations, 2) start-up cost (7:216).

Early object-oriented programming languages such as Smalltalk were interpreted and thus inefficient compared to a conventional programming language (32:10). Subsequently, performance sensitive systems could not be designed and coded in object-oriented languages. Today, with the introduction of new languages, OOD systems have improved in performance. Initial investments in education, tool support, reorganization, and management support are necessary in order to eventually realize the benefits of OOD.

2.3 Reuse

Reuse is the use of previously acquired concepts (the reuse of ideas and knowledge) and objects (the reuse of particular artifacts and components) in a new situation. It is the process of building software systems from existing software rather than building software systems from scratch. Very significant process has been made in the evolving field of software reuse.

The main motivation to reuse software artifacts is to increase software development and maintenance productivity; this leads to higher quality, more reliable software, and conservation and preservation of software engineering expertise (14:3). Portability is a characteristic of software closely related to reusability. It refers to the extent to which a software component can be used in multiple machine environments. Thus reusability includes portability in the sense that portability is necessary to achieve reusability across multiple machine environments at least at code level.

The U.S. DoD software engineering community is in its pursuit of software reuse, and has seen evidence that the software reuse principle, when integrated into acquisition practices and software engineering processes, provides a basis for dramatic improvement in the way software intensive systems are developed and supported over their lifecycle. Availability of the Ada language has spurred interest in reuse, and Ada serves as the implementation language in many reuse projects. At the highest level, the DoD vision for reuse is to drive the DoD software community from its current "re-invent the software" cycle to a process-driven, domain-specific, architecture-centric, library-based way of constructing software. The DoD's long-term strategy is to lead to the creation of a true "black-box" components industry (28).

There is great diversity in the software engineering technologies that involve some form of software reuse. Typically, reuse involves the abstraction, selection, specialization, and integration of artifacts, although different reuse techniques may emphasize or de-emphasize certain of these. Krueger partitioned the different approaches to software reuse into eight categories: high-level languages, design and code scavenging, source code components, software schema, application generators, very high level languages, transformation systems, and software architecture, analyzed them according to the following taxonomy (22:137).

- Abstraction: What type of software artifacts are reused and what abstractions are used to describe the artifacts?
- Selection: How are reusable artifacts selected for reuse?
- Specialization : How are generalized artifacts specialized for reuse?
- Integration: How are reusable artifacts integrated to create a complete software system?

2.3.1 Reuse in Design and Code Scavenging. The reusable artifacts in scavenging are source code fragments. The abstractions for these artifacts are informal concepts that a software developer has learned from previous experience. When a software developer recognizes that some part of a new application is similar to one previously written, a search for existing code may lead to code fragments that can scavenged. Specialization of a scavenged code may be done through manual editing. Integrating a scavenged code into a new context may require some modification of the fragment, the context, or both. In ideal cases of scavenging, large fragments of source code can be adapted without significant modification. But in the worst case, lots of time can be wasted understanding, modifying, and debugging a scavenged code rather than developing the equivalent software from scratch.

2.3.2 Reuse in Source Code Components. Currently, the best abstractions for reusable components are domain-specific concepts, such as those found in the math libraries. The area of code reuse, including deliverable code, test code, simulation code, or etc. is the highest potential for near-term payoff. There are two different categories of code components for reuse - "passive components" or "building blocks", which are used essentially unchanged, and "dynamic components" or "generator", which generate a product for reuse (18:83-84). Although selection in domain-specific components is easy since components can be classified, organized, and retrieved using well defined properties of the domain, the ease of selection in a general-purpose component library depends on the degree of the abstraction, classification, and retrieval schemes. Generalized components with construction-time parameters represent the most effective approach to component specification. Reusable components can be specialized either by editing the source code directly or with mechanisms such as the Ada generic or inheritance in object-oriented languages. Ada generics provide a level of abstraction that isolates the software developer from many implementation details. Ada generics can be parameterized with language constructs such as data types, data objects, and functions. Module interconnection languages such as Ada typically provide the framework for integrating components. Ada can integrate source code components written in C, FORTRAN, and assembly. Naming and name binding are important module interconnection issues in component reuse since reusable components are constructed independently of any particular context, which can present problems such that names imported into and exported from the component may crash or be incorrectly bound in the new system. Ada is a particularly good candidate for implementing reusable components since Ada provides mechanisms to overcome some of these naming problems.

2.3.3 Reuse in Software Schemas. The schema approach emphasizes the reuse of algorithms, abstract data types, and higher level abstractions. It is a formal extension to reusable software components. For example, some algorithm books provide a library of abstract descriptions for many basic computer science algorithms and data structures. Programmers can informally use these abstractions when writing source code. Large schema libraries are difficult to use; however, automated assistance can help for schema selection. Schemas are typically specialized either by substituting language constructs, code fragments, or specification into parameterized parts of a schema or by choosing from a predefined enumeration of options. A simple approach to schema integration is to use a module interconnection language. For example, if a schema instantiation produces Ada package code, an instantiated schema can be treated as a conventional Ada package. More sophisticated schema integration techniques rely on semantic specifications.

2.3.4 Reuse in Software Architecture. Software architectures are analogous to very large-scale software schemas. Software architectures, however, focus on subsystems and their interaction rather than data structures and algorithms.

2.3.5 Reusability in an Object-Oriented Approach. The object-oriented paradigm combines design techniques and language features to provide strong support for reuse of software modules. Reuse comes in a variety of forms. Some of the reuse in the object-oriented paradigm is much the same as that in the procedural paradigm, but the object-oriented paradigm adds an additional type of reuse.

Every time an instance of a class is created, reuse occurs. This is similar to the declaration of a variable of a specific type. The major difference is that the resulting class instance is a much more complex structure than a simple variable. An instance of a class provides a combination of data structures and operators on those data structures. This is similar to (but more general than) library reuse with the conventional paradigm.

Inheritance provides levels of support for reuse (21). As part of the high-level design phase, inheritance serves as a means of modeling generalization/specialization relationships. These relationships appear in the form of classifications. A chair may be viewed as a special type of furniture, as well as a general description of the more specific categories of rocking chairs, straight chairs, and reclining chairs. This high-level use of inheritance encourages the development of meaningful abstractions which, in turn, encourages reuse.

Often in actual design, the presence of mid-level abstractions, such as table and chair, will be recognized and considered separately. The availability of an inheritance relation enables the designer to "push higher", to identify commonalty among abstractions, and to produce higher-level abstractions (e.g., furniture) from this commonality. By identifying this commonality and moving it to a higher abstraction, it becomes available to be reused later in the current design or in future designs. Filing cabinets and bookcases may be identified later. Much of their description (attributes such as height, weight, color, etc.) may already be available from the furniture abstraction. The benefits of this reuse prompt the designer to search for higher and higher levels of abstraction.

In the low-level design phase, inheritance supports the reuse of an existing class as the basis for the definition of a new class. An existing piece of code can be copied to a new file and modified to fit its new purpose. This inheritance mechanism does not establish any connection between the old piece of code and the new code.

For example, algorithm reuse involves using the same algorithm across data structures. Using the data abstraction supported by object-oriented technology, such an algorithm is implemented at a high level of a class hierarchy and becomes automatically available to subclasses.

These mechanisms promote reuse by means of *interface abstraction.* An interface specification is the most abstract reusable artifact of a software system. It consists of a set of messages that embody a coordinated set of behaviors. Classes whose instances perform the role implied by these behaviors must provide a behavior implementation. These instances can then be used wherever this common behavior is expected.

2.4 The Features of Ada

This section provides a quick overview of the advanced features of the Ada language. The DoD designed Ada as a general-purpose language intended to embody and enforce the principles of software engineering in hopes of lowering the cost of the software life-cycle. Ada's objectives were not to extend the realm of things that computers can do, but to provide a single way to do the things that are now done in numerous incompatible but similar languages (13:38). "Ada is a design language that is suitable for the design and implementation phases of the software life cycle. Ada directly embodies, encourages, and enforces modern software engineering principles and methodologies" (30:163).

2.4.1 The Object Oriented Capabilities of Ada. Ada has been traditionally associated with Object-Oriented Design (OOD), which was exploited by Booch (7). However, OOD can be extended more easily and smoothly through Object-Oriented Programming (OOP), which has basically two additional features, *inheritance* and *polymorphism*, that cannot be fully extended by object-based programming language such as Ada. Ada's support for the two additional features is less systematic than that found in C++, which fully supports these features. Alternatively, one might think of OOP in terms of two programming paradigms, which will be associated with OOP:

- variant programming: new abstracts may be constructed from existing ones so that the programmer need only specify the differences between the new and old abstractions.

- class-wide programming: classes of related abstractions may be handled in a unified fashion, such that the programmer may systematically ignore their differences when appropriate (1).

Ada provides the basis for supporting object-oriented programming with variant and class-wide programming in the form of derived types, subtypes, packages, and generic units. Each of these has its limitations such as type incompatibility by deriving from the parent types, narrowing their applicability by subtyping, recompilation of the original abstraction by breaking the original abstraction, and complicated generic parameters. Ada supports several forms of static polymorphism: generic formal types, subprogramming overloading, and implicit conversion of class-wide (real and integer) literals.

Recognizing limitations, the Ada 9X program is preparing a refined version of Ada to update the Ada standard in accordance with ANSI and ISO procedures under the Ada Joint Program Office (AJPO). The current Ada 9X review process is adding changes to improve Ada's use in OO development, programming-in-the-large and real-time programming. Ada 9X will provide improved support for OO development in several ways (1):

- subprograms as objects: dynamically selecting subprograms
- reducing the need for recompilation
- programming by specialization/extension: defining a new entity which can be used anywhere the original one could be, in exactly the same way without modification of the original one.

Subprograms as objects provides the ability to associate operations (subprograms) with objects, and to dynamically select and execute those operations, which is the basis for run-time polymorphism. Programming by specialization/extension and reducing the need for recompilation provides the ability to extend derived reuse, which is the basis for the inheritance mechanism.

2.4.2 The Reusability of Ada. The Ada programming language has several mechanisms which aid in the specification and development of reusable code. (19:486)

- Ada supports reusability through the package concept. This is a mechanism whereby one can define an Abstract Data Type (ADT), and it supports the notion of information hiding.
- Ada supports reusability through generics (generic procedures in Ada are mechanisms which preserve the virtues of typing, while eliminating the negative aspects). Generics can be procedures or packages - but reusable generic packages are a more powerful concept than that of procedures.

2.4.3 Foreign Language Interface. One unique feature of Ada is its ability to interface to other languages. The *interface* pragma allows an Ada program to call a program written in another language such as C, FORTRAN or assembly language. An Ada program calling a subprogram written in another language must include a declaration for that subprogram, written in the usual Ada notation for subprogram declarations. The actual code in the other language plays the role of an Ada subprogram body, so the Ada program includes an *interface pragma* instead of the actual subprogram body (11:807). An implementation may restrict the use of the *interface pragma*. For example, it might establish a correspondence between certain predefined Ada types and types in the other language and require that each subprogram parameter and function result belong to one of these types.

2.5 The Features of C++

C++ is a strongly-typed language developed by Bjarne Stroustrup at AT&T Bell Laboratories as an extension of C. The primary difference between C++ and C is the support C++ provides for the following: (23:580)

- Inlining and overloading of functions.
- Ability to provide default argument values.

- Argument pass-by-reference (in addition to the C language default pass-by-value).
- Support of template functions and classes.
- Support of abstract data types by providing for information hiding and the definition of a public interface.
- Support for object-oriented programming.

The language supports object-oriented concepts such as abstract data type, inheritance, polymorphism, and dynamic binding (20:396).

- *Abstract Data Type*: C++ provides two constructs for defining an ADT. The first one is an extension of the *struct* construct and the other is the *class* construct.
- *Inheritance*: C++ allows the hierarchy of class definitions to inherit both method and instance variables from existing class definitions.
- *Overloading/Overriding and Dynamic Binding*: C++ allows overloading of function names and operators. It allows single polymorphism but not parametric polymorphism (or genericity) which is supported by many object-oriented languages and also Ada. C++ also supports dynamic binding through virtual functions.

2.6 Ada/C++ Similarities and Contrasts

Programming language selection is not the major cost driver in a software development environment (14). But languages facilitating software engineering methods and principles can produce software easier to learn and understand, easier to reuse, easier to change and maintain, and easier to interface with other languages and CASE tools. Both Ada and C++ are general-purpose languages of roughly similar power and have features that modern software engineering practice considers indispensable: modularity, information hiding, abstraction, structuring tools for large programs, and various mechanisms for parameterizing software components. The following comparison and contrasts between Ada and C++ is based on current versions of the languages. For Ada, the language is defined by ANSI/MIL-

STD-1815A-1983. For C++, the language is defined by version 2.1 of the AT&T CFRONT translator.

Comparing Ada and C++ is not easy, if for no other reason than C++ is a language in flux for which there is no stable definition, no approved standard reference, and no translator validation.

C++ requires more knowledge than Ada, but this knowledge is ill-defined at the interface between environment and language. This reduces portability and thus increases maintenance costs more than comparable Ada software. C++ software is less reliable than Ada since arrays in C++ are closely related to pointers, and the indexing operation is described directly in terms of pointer arithmetic. The generic facility of Ada is an excellent model of type parameterization to maximize software reuse rather than C++, although C++ provides a template which is close in spirit to Ada generics. C++ emphasizes ease of writing rather than ease of reading. This makes C++ programs harder to transmit and maintain (33:15). Ada has demonstrated maintainability and reliability for large-scale development. Ada is safer but less flexible than C++. Currently, Ada has not been used extensively in several key areas. A couple of important changes planned include extending Ada's data abstraction capabilities, adding object-oriented programming features, and improving control over concurrency for real-time applications (1).

C++ is already a widely accepted object-oriented language in the commercial area and is becoming even more popular since it has a C and Unix base. C++ is highly flexible and therefore less safe than Ada. The emerging C++ standards will help to increase portability and maintainability of C++.

Table 2.1 shows some important language features and their relative support by the two languages (6:2-9). Interfacing well with other languages is an important attribute of any programming language. The C++ language can invoke directly C run time libraries and existing C scftware with C interfaces. Ada defines an optional pragma interface for interfacing to other languages. Some advantages depend on their compiler support.

Feature	Ada(only)	Ada(+)	Both(=)	C++(+)	C++(only)
Parameterized Types	X				
Safe Types	X				
Error Handling	X				
Concurrency	X				
External Interrupts	X				
Compilation Management		X			
Strong Types		X			
Modularity			X		
External I/O			X		
Extensible Typing			X		
Overloading			X		
Multilingual Support			X		
Polymorphism					X
Inheritance					X
Subprogram Variables					X
Conditional Compilation					X

Table 2.1. Ada and C++ Support for Key Language Features

2.6.1 Features Where Ada has an Advantage. The following paragraphs discusses the features where the Ada language has an advantage over the C++ language (6:2-10).

- Parameterized Types : A parameterized mechanism is useful for building strongly typed reusable component libraries. Ada provides this useful support through generics. Although some users of the present versions of the C++ language provide their own template preprocessors for this feature, it is not available commercially in C++.
- Safe Types : Ada provides run-time checks, array subscript variables and ranges. C++ does not provide bounds checking. C++ provides flexible dynamic memory allocation which must be used carefully to prevent problems.
- Error Handling : For reliable and maintainable systems, a reliable standard mechanism for handling errors is essential. Ada provides user defined exceptions and some useful predefined exceptions for error handling.

- Concurrency : Ada provides support for concurrency with tasks for the efficient implementation of a large system.
- External Interrupts : Ada provides a standard mechanism for handling interrupts from the external environment as task entry points.
- Compilation Management : Efficient management of compilation dependencies and good compilation dependency information can not only save large amounts of computer and human resources but also simplify the creation of software tools such as configuration management tools, test generator and code analysis tools. These compilation dependencies are well defined in Ada, but not as well defined in C++.

2.6.2 Features Where C++ has an Advantage. Listed below are the features where C++ language has an advantage over the Ada language. (6:2-10)

- Inheritance : C++ supports both single and multiple inheritance. This feature is not available in Ada. The C++ inheritance is more powerful than the derived typed mechanism in Ada. Ada inheritance is expected to appear in the Ada 9X language.
- Polymorphism Languages : C++ supports polymorphism through its inheritance mechanism. Inheritance and polymorphism are expected in Ada 9X.
- Subprogram Variables : C++ has pointers to functions. Pointers to subprograms are expected in Ada 9X.
- Conditional Compilation : C++ supports conditional compilation, via the preprocessor mechanism.

2.7 Ada Interfacing (Binding) with C++/C

In general, translating programs from other languages into Ada is straightforward if the source language is one of the block-oriented languages such as C. However, it is more desirable to make use of existing subprograms or libraries developed in some other language from inside Ada programs without having to translate everything into Ada. There is a need in development of software systems for utilization of capabilities which are not inherently supported by a chosen development language. For example, if a graphics capability was required, the programmer typically used a specific vendor's graphics interface. In order to overcome this deficiency in a cost-effective manner, bindings are required. An interfacing (binding) is a set of code which allows the use of software and hardware that provides some capability required for a given application. In the past, software systems tended to be built around a specific product, which decreased portability. Standard interfaces were required to provide portability. Ada provides mechanisms that allow the programmer to specify interfacing which is no longer limited to a single machine. This section presents the naming convention, parameter passing method, and an approach to making existing libraries and programs written in C++/C usable from Ada.

2.7.1 Ada Interface. Ada has a complicated naming convention for its objects that can be accessed from another language. For example, subprogram names are usually encoded as follows: (12:21-25)

_A_subprogramnameLLXcc.parent

where:

subprogramname is the subprogram name.
LL is the line number of its definition.
X is S if defined in the spec and B for the body.
cc is the character number of its definition.
parent is the name of the parent unit, without the _A_ prefix.

To access a Ada object, a user must be able to modify the Ada source code to know where the subprogram is declared. Another big problem with the Ada naming convention is that when you change the location of the subprogram in its source program, the external name changes. Fortunately, Ada provides the *pragma* that you can use to specify an unchanging external symbol name for variable and function. These pragmas are *external_name*, *interface*, and *interface_name*. The *pragma EXTERNAL_NAME* allows users to specify an external symbol name, or linkage name, for Ada variables or subprograms so that it can be accessed from another language. The *pragma INTERFACE* allows users to call routines written in another language. The *pragma INTERFACE_NAME* allows users to access external objects such as common blocks, C global variables, and routines written in another language.

Ada allows data to be passed by reference or by value using the formal argument mode. To pass a parameter by reference users must use the *in out* parameter mode. To pass a parameter by value users must use the *in* mode. However, arrays and records must always be passed by address.

There are a couple of restrictions with the *pragma interface* (12:24).

- The types of parameters for C routines must be *scalar, access* or the predefined type *address*, and all parameters must have mode *in.*
- The return types are limited to *scalar, access* or the predefined type *address.*

2.7.2 C Interface. C has the single naming convention for external symbols, which include function names and global variables. The C compiler prepends an underscore character ('_') to external symbols. Additionally, function names and global variables produced by the C compiler are unrestricted in length and case sensitive (12:15-16). For example, the external symbols produced for the following code fragment are *_add* and *_num3.*

```
int add( int num1, int num2)
{
   extern int num3;
   return (num1 + num2 + num3);
}
```

C functions basically pass all parameters except arrays by value, which means that only the contents of a parameter are passed to the called routine, not its address. Arrays are passed by reference. However, C provides two operators users can use to work around this parameter passing method: address operator (&) and indirection operator (*).

2.7.3 C++ Interface. Like Ada, C++ generates symbols that do not simply have underscores prepended and appended to a function name. Although a data types and function parameter passing method in C++ are basically the same, function symbol names usually have a return type and parameter data types encoded in them. Fortunately, C++ provides a *linkage specifier* that causes C++ to generate a symbol name that conforms to the C language interface. The AT&T C++ Language System Product Reference Manual describes the following information about the *linkage specifier* (3:40). Linkage to non-C++ code fragments can be achieved using the following *linkage specifier*:

extern *string-literal declaration*
extern *string-literal* {*declaration-list*}
where:
string-literal can be "C" or "C++" to indicate whether a declaration should have C or C++ linkage. Default is C++
declaration is a function or variable declaration
declaration-list is a list of function and variable declarations

Linkage specifications nest. A linkage specification does not establish a scope. A *linkage-specification* may only occur in file scope. A *linkage-specification* for a *class* applies to non-member functions first declared within it. A *linkage-specification* for a function also applies to functions declared within it. A linkage declaration with a string that is unknown to the implementation is an error.

2.7.4 Ada Binding to C++/C Routines. Use of the Ada language facilitates portability, as compared to other languages. The features of the Ada language support portability through *abstraction* and *information hiding.* Ada packaging allows the encapsulation of both data and operations into a single unit, the enforcement of strong typing and information hiding, the separation of the specification and body, and the isolation of the system dependent features. The Ada package for calling the library functions or existing code gives the programmer basically the same functional entities and objects as the original.

There is a series of procedures which are required to successfully develop a binding. For a complete Ada interface to a library and existing codes using the same subprogram and variable names provided in the original C++/C version, the following six steps are necessary (34:156):

- Create parallel data types.
- Interface to external routines.
- Interface to external data.
- Link to external libraries.
- Test/Debug the interface.
- Optimize the interface.

2.7.4.1 Create parallel data types. Whenever access to a routine or variable declared in an alternative language is required, any Ada variable used in conjunction with the subroutine or variable is of a compatible data representation in both languages. When creating Ada data types to parallel the types of other languages, the user should not assume that the types or structures have the same implementation in Ada, even if they have the same name; that is, a data structure declared in Ada must be identical to a data structure declared in C++/C.

A way of creating parallel data types is to use a priori knowledge. There are some types that the programmer knows are parallel between two language implementations from reading the vendor's documentation. Neither Ada nor C++/C compilers are required to use a particular size to represent any particular type, and an implementation is free to choose a representation based on hardware considerations.

Another way of creating parallel data types is to use Ada representation specifications. In Ada, we can define an exact duplicate of the physical layout of any data type in another language once it is known. It can be done by Ada representation clauses. When the underlying representation of a type has no analogue in one language, the data type can be defined by the programmer using Ada representation specifications and *UNCHECKED_CONVERSIONs*.

The conversion of six different type categories can be described as follows. (41:PG4-2).

- Simple Types: There are some simple Ada predefined types that correspond to C simple types. When C++/C programs contain ambiguous assignments or uses of such types or of integer/address conversion, the generic function *UNCHECKED_CONVERSION* offers a method for controlled easing of type conversion. For example, to implement an Ada type to match a C++/C *int* type, a programmer could specify as follows:

  - type C_int is range -(2**15) .. (2**15) - 1;
  - for C_int'size use 16;
  - C_int_use : C_int;

  The first line represents a type that has the same range as the *int* type in C++/C. The second line ensures that the same amount of storage is used. The third line declares a variable. Other simple types with different representations can be constructed similarly.

- Record Types: The same basic approach can be taken in the representation of record types as with simple types. Both Ada and C++/C associate the record label with a base address from which offsets to access individual components of records are calculated. In Ada, as long as the record is composed of equivalent simple data types, the offsets will be calculated similarly, and record structures will be identical. When storage conventions are not so conveniently arranged, Ada representation specification can be used for constructing records.

- Array Types: Ada and C++/C arrays are stored in row-major order. When defining Ada array types that are parallel to C++/C array types, the standard representation of an array in both languages is to associate the array label with the first component and use this location to calculate an offset. The individual components should be compatible structures. Otherwise, representation specifications should be used to assure that indiv' al component representations are identical.

- Dynamic Array Types: In C++/C, the size of a dynamic array is calculated by the user, based on data known only to the user. C++/C arrays always start at index 0 while Ada arrays start with any index. Passing arrays from C++/C to Ada is possible by creating an appropriate subtype for the value. If a C++/C array is passed and must be preserved over an open scope, a fixed-length array must be used in the Ada program, making the Ada array at least as large as any possible C++/C parameter.
- Pointers and Address Types: Pointer and address types are implementation-specific. But Ada's tactic of using host conventions usually allows the use of Ada pointer and address types parallel to their C++/C counterparts. Otherwise Ada representation specifications can be used to tailor the size and range of the data type.
- String Types : A character string in C++/C is represented by a pointer to the first character in an array of bytes. By convention, strings in C++/C are terminated by a null character and store no explicit length. In Ada however, a string is represented by a packed array of type *CHARACTER* with the maximum number of components specified as part of type. A parallel type can be created using a declaration in Ada as follows:
  - type C_String is access STRING (1..INTEGER'LAST)

2.7.4.2 Interface to external routines. Once parallel types have been established, the next step is to gain access to external routines provided in the interface target package. This is accomplished in a two stage procedure: first, equivalent Ada subprogram specifications are written, and second, the linkage to the external routine is declared. The first step can be a simple mapping of the external routine's name and parameters into an Ada subprogram specification or can involve the development of code to make the behavior of the external routine compatible with Ada. The second step is accomplished through use of the *pragma INTERFACE* and *pragma INTERFACE_NAME*.

2.7.4.3 Interface to external data. The third step when building a complete package from an existing C++/C library and program is to gain access to external variables declared in C++/C from Ada. Some Ada compilers contain a *pragma* which allows the accessing of external objects directly, while others require the programmer to build an external routine which returns the required data object as a parameter (34:159).

For example, the following programs illustrate interfacing between Ada and C.

C program:

```
  ...
char *gets ();
int atoi ();
int service_number;
extern void ada_put ();
test ()
{
  char buf[80];
  printf(" Enter an integer here: ");
  get (buf);
  service_number = atoi(buf);
  ada_put(service_number);
}
waddch(window win,char [])
{
  .....
}
```

The *printf* call was replaced with ADA_PUT and an Ada package containing the procedure ADA_PUT and interface declarations for the C entities were written as follows.

```
  ....
with language;
package C_interface is
        .......
        service_number : integer;
        pragma interface_name(service_number,C_SUBP_Prefix & "service_number");

        procedure waddstar(win : window; S : address);
        procedure c_waddstr (win : window; str : address);
        pragma interface( C, C_waddstr);
        pragma interface_name(C_waddstr, C_SUBP_PREFIX & "waddstr");
        procedure ADA_PUT(I : integer);
        pragma external (C, ADA_PUT);
        pragma external_name(ADA_PUT, C_PREFIX & "ada_put");

        procedure main;
        pragma interface(C,main);
        pragma interface_name(main, C_SUBP_PREFIX & "test");
        .......
end C_interface;
package body C_interface is
        .......
        -this intermediate Ada module will convert an Ada string
        -input into a c-string format before calling the C routine
procedure waddstr(win : window; S: string) is
        T: string(1..(S'last + 1));
begin
        T(1..S'last) := S;
        T(S'last + 1) := ascii.nul;
        c_waddstr(win, T'address);
end waddstr;
procedure ADA_PUT (I : integer) is
begin
        put(I);
end ADA_PUT;
end C_interface;
        ....
```

Now a simple Ada "wrapper" to call the original C functions is written so that the linker *a.ld* can resolve all the references in the modules and perform its usual elaboration order checks.

```
with C_interface; use C_interface;
procedure driver is
        win : window;
begin
        main;
        waddstr(win,"hellow");
        ....
end Driver;
```

2.7.4.4 Link to External Libraries. The fourth step in the Ada interface is the ability to link the routines and data types built in the previous steps with the external libraries. It is typically required to place the names of the external object files in the link path. For example, illustrated programs compiling both the C and Ada portions can be done by compiling the C portion first, then using the Ada linker to construct the 'main' program *driver* and the C objects in the link.

2.7.4.5 Test/Debug the Interface. The fifth step is testing and debugging the binding. Since a good binding will be used by a wide variety of software systems, the testing and debugging should be done thoroughly in order to construct a reliable system.

2.7.4.6 Optimize the Interface. The final step in the interface process is to reduce the overhead resulting from frequent subprogram calls to intermediate routines written in Ada. The predefined *pragma INLINE* provides the solution.

2.7.5 Program Conversion. One modular approach is to write an Ada "wrapper" program that surrounds the subprograms in another language and allows them to be gradually converted (41:PG4-12). *Pragma INTERFACE* and *pragma INTERFACE_NAME* are useful for this gradual replacement with *pragma EXTERNAL* and *pargma EXTERNAL_NAME* that allow subprograms in other languages to call Ada subprograms, exactly the reverse of the *INTERFACE* and *INTERFACE_NAME* pragmas.

The real benefit for the user is that new portions of large programs can be developed in Ada, but existing, tested, working code need not be replaced wholesale. Individual modules can be replaced by newly developed Ada code without undue restrictions on the language of calling or called subprograms. An additional benefit is that once subprogram parameters are defined in Ada, the compiler will perform its usual type checking across subprograms.

2.8 The MICROSTICK

The MICROSTICK is a professional high quality point and select device. One of its standard features includes a resolution of 1 part in 4096 and six types of movement. These features allow users to change gears and use the MICROSTICK for both high-resolution accuracy and fast pointing speed. The MICROSTICK is connected to a computer or a terminal via a 25-pin RS232 connector. The MICROSTICK outputs an 18 byte string of ASCII characters. The outputs of MICROSTICK are described in the table 2.2. Byte 1 corresponds to a delimiter, Byte 2 describes the state of button 2, etc.

Byte	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
What	Delimiter	Button 1		Button 2		Button 3
Example	$	0		1		0
Byte	Byte 7	Byte 8-11	Byte 12	Byte 13-16	Byte 17-18	
What		x value		y value	Delimiter	
Example		1513		0028	[cr][lf]	

Table 2.2. Outputs of MICROSTICK

This output allows easy interface to a microcomputer or a terminal. In addition, the MICROSTICK microprocessor-based design allows for user-specified models of operations that permit easy adaptation to most applications. The user's manual describes the MICROSTICK and suggests ways to realize its full potential.

The MICROSTICK can be controlled by Graphics, CAD/CAM or Text Editing software. Hence, software controlling the MICROSTICK is a good candidate for use in this research.

2.9 Summary

This chapter has introduced the features of the object-oriented paradigm, software reuse, the features of Ada and C++ languages, and the MICROSTICK.

The object-oriented paradigm represents a more intuitive way to program than using procedurally oriented techniques. The object-oriented approach is based upon the concepts:

objects, classes, abstraction, inheritance, encapsulation, polymorphism, and dynamic binding. The object-oriented paradigm offers a way to manage the complexity inherent in a software system, and supports the goals and principles of software engineering.

Reuse is the use of previously acquired concepts (the reuse of ideas and knowledge) and objects (the reuse of particular artifacts and components) in a new situation. It is the process of building software systems from existing software rather than building software systems from scratch. The main motivation to reuse software artifacts is to increase software development and maintenance productivity in order to obtain higher quality, more reliable software, and conserve and preserve software engineering expertise. There is a great diversity in the software engineering technologies that involve some form of software reuse. Typically, reuse involves the abstraction, selection, specialization, and integration of artifacts, although different reuse techniques may emphasize or de-emphasize some of these.

Programming language selection is not the major cost driver in a software development environment. But a language facilitating software engineering methods and principles can produce software easier to learn and understand, easier to reuse, easier to change and maintain, and easier to interface with other languages and CASE tools. Both Ada and C++ are general-purpose languages of roughly similar power. Both have features that modern software engineering practice considers indispensable such as modularity, information hiding, abstraction, structuring tools for large programs, and various mechanisms for parametrizing software components. C++ requires more knowledge than Ada, but this knowledge is ill-defined at the interface between environment and language. This reduces portability and thus increases maintenance costs over comparable Ada software. C++ software is less reliable than Ada since arrays in C++ are closely related to pointers, and the indexing operation is described directly in terms of pointer arithmetic. The generic facility of Ada is an excellent model of type parameterization to maximize software reuse, although C++ provides the template which is close in spirit to Ada generics. C++ emphasizes ease of writing rather than ease of reading. This makes C++ programs harder to transmit and maintain. Ada is safer but less flexible than C++.

Translating programs from other languages into Ada is straightforward if the source language is one of the block-oriented languages such as C. However, it is more desirable to make use of subprograms or libraries developed in some other language from inside Ada programs without having to translate everything into Ada. For a complete Ada interface to another language, the following six steps are necessary

- Create Ada data types.
- Interface to external routines.
- Interface to external data.
- Link to external libraries.
- Test/debug the interface.
- Optimize the interface.

The MICROSTICK is a professional high quality point and select device. The device allows users to change gears and use the MICROSTICK for both high-resolution accuracy and fast pointing speed. Connecting is done through a 25-pin RS232 connector. The MICROSTICK outputs an 18 byte string of ASCII characters. Software controlling the MICROSTICK will be used in this research.

III. Analysis/Design

3.1 Introduction

This chapter describes the analysis and design phase of this thesis work. The previous research efforts by other AFIT students have generated many objects. Each of the objects can be reused, but the reuse process may differ depending on the representation of each of the objects and on the effort to reuse. This chapter is concerned with derived reuse that is accomplished via the OO principle of classes/objects and their relationships. As a part of this, several ways of building Ada software components are discussed. The discussion will range from domain analysis and component identification to the development of effective reusable software components for flight simulator applications. The concerns are the characteristics of good reusable software components, such as maintainability, understandability, ease of use, the importance of quality, and generality.

When building a reusable software component, a systematic approach to identify and to develop reusable components is needed. This usually comes from domain analysis that leads to the identification of common obje:ts, operations, and structures. Reusable code components are designed with the following goals in mind:

- Reusability: The design should provide for reusing existing code and a framework that enables the reuse of new code.
- Extensibility: The design should be constructed so that future additions to the design can be made with a minimum effort.
- Utility: Design of each component should include useful, easy, and flexible object functions.

3.2 Analysis

While the focus of this thesis is the design and implementation of a reusable component to evaluate the features of Ada, it is not possible to simply start with a design. The analysis should be done first in order to have a starting point for the design. Analysis is concerned with devising a precise, concise, understandable, and correct model of the real world. An analysis model is built to abstract essential aspects of the application domain without regard for eventual implementation. This model contains objects found in the application, including a description of the properties of the objects and their behavior.

As a part of the analysis, the understanding of the system and object notations used in this thesis was required. This came from Captain Simpson's system and object notations (37). The analysis started with domain analysis of the flight simulator.

A domain analysis is an investigation of a specific domain or application area to identify a common "generic paradigm" and to identify candidate reusable components for the domain (14:13). A domain analysis is similar to a system analysis, but is much broader in scope. The domain analysis results in development of a domain model that provides the framework for development of reusable software components. In other words, it leads to the identification of common objects, operations, and structures. One of the most important things in performing a commonality study is a level of commonality, which is captured by means of abstraction/decomposition, generalization/specialization and parameterization.

There are several techniques that can be used in performing a commonality study. One of them is an OOA software decomposition technique based on the classes of objects, which are viewed as a "high-level abstraction", as in an ADT. An ADT is a class of objects defined by a set of operations available on them and the abstract properties of these operations. An OOA of the domain and each member of the domain representation set can lead to the identification of commonality across applications, and can be used as a good starting point for the development of a domain model and associated reusable software components. A class in an object model, as Booch points out, is a set of objects that have a common structure and common behavior. The object class is a candidate reusable component for the domain. Therefore an OOA of the domain can serve as domain analysis for building a reusable software component.

The main purpose of the analysis phase was to identify potential reusable software components. The identification of reusable components was based mainly on the flight simulator object model shown in Appendix A (37). Several base classes were identified as reusable components, then the low level inputs (Joystick and RS232 Port classes) were selected as reusable components for use in this thesis. The Joystick and RS232 port are not typical graphics components, but they work much like graphics components. The system interface to control the I/O devices is essentially required to use system calls written in C much like Graphics Library interfaces. Hence, software components controlling the low level devices such as I/O and Graphics devices are good candidates for reusable software components, and they illustrate the same principles as we would find in building reusable graphics components.

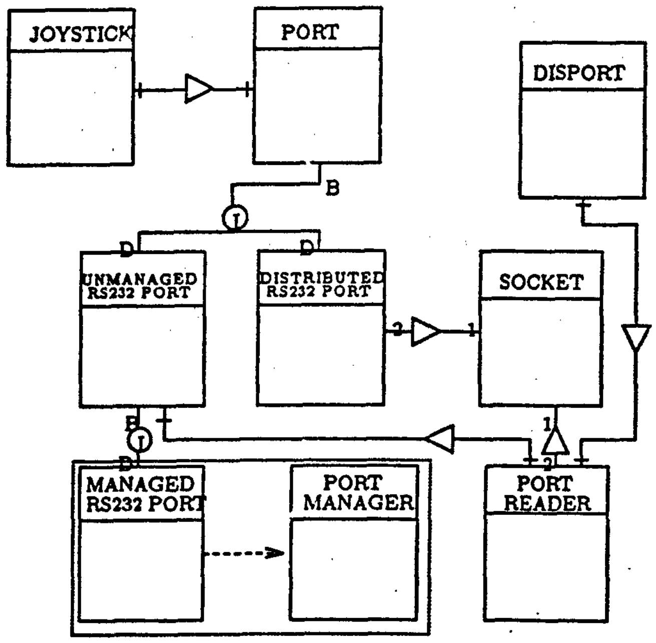

Figure 3.1. Design of Joystick and RS232 Port Classes

The next step was to analyze the joystick class and the RS232 port classes and their relationship among their class members in more detail. The detailed analysis was based mainly upon the sources available. The main available sources of information for analysis were the user's manual for the joystick, previously written C++ routines, and the object-oriented model. The object-oriented model of the Joystick and RS232 port objects are shown in the figure 3.1. They served as a basic domain model for building reusable components in this thesis.

The joystick communicates with the computer through an RS232 port. The RS232 port was accessed through Unix system calls. The Joystick used both the Managed RS232 Port and the Distributed RS232 Port. The Distributed RS232 Port provides a transparent interface with another machine through the Unix socket. Since both the Unmanaged RS232 Port and the Distributed RS232 Port have common methods, an abstract class named "Port" was created. The "Disport" object is the "main" program, which makes and runs the Port Reader. The Port Manager object is responsible for the extra checking of port usage on different machines.

The next step was to identify existing code that could be reused in the design and implementation in order to reduce the overall effort required. The reused code came from the Unix system library and the flight simulator class library. For example, the Unix system library is required to control a hardware device. But reusable code was written in C++ or C and was not visible directly from Ada. For a successful Ada binding to C++ and C, the binding feasibility was analyzed. There is a series of procedures that is required to successfully develop a binding. For a complete Ada binding to a library and existing code using the same subprogram and variable names provided in the original C++ or C version, the following steps are necessary.

- Create parallel data types.
- Interface to external symbols (routine and data).
- Link to external libraries.

Whenever access to routines or variables declared in an alternative language is required, any Ada variable used in conjunction with the subroutine or variable is compatible with the C++ or C data representation. Once these parallel types have been established, the next step is to gain access to external routines and data provided in the interface target language. This is accomplished in a two stage procedure: first, equivalent Ada subprogram specifications are written, and second, the linkage to the external routine and data is declared. The first step can be a simple mapping of external routine and data names and parameters into an Ada subprogram specification. The second step is accomplished through use of the *pragma INTERFACE* and *pragma INTERFACE_NAME*. Linking the symbols built in Ada with the external C++ or C symbols is done by the Ada linker. The detailed binding process is presented in chapter 2.

For each class object, a detailed analysis was performed. Basically, the Unmanaged_RS232 Port class was a wrapper class to call Unix system calls to control an RS232 port for the input on one machine. Captain Simpson's code was instrumental in its detailed Unix system calls to control the RS232 port (37). Using Captain Simpson's code and a variety of Unix system calls (36), all information necessary for analyzing a wrapper class was obtained. The primary motivation for making a wrapper class is to make library or operating system routines easier to use. Thus, only the file descriptor was needed once the RS232 Port was opened. Most of the complexity occurred in the initialization of the port. Given the Unmanaged_RS232 Port, this work shifted into how to reuse existing C code within Ada. Most of the Unix system calls in the Unmanaged_RS232 Port class were used to control I/O devices, and some had complicated data structures. Thus, successful development of a binding (interfacing) to Unix system calls was required. Fortunately, VADS provided the 'graphiclib' and 'publiclib' libraries which provide ways to interface with Unix system calls (41). These libraries provided parallel data structures between Ada and C corresponding to most of the system calls for I/O and graphic devices. For example, the *IOCTL* system call was needed to represent the way the port should function, such as enable receiver, enable signals, and enable user specified baud rate. VADS provided exactly the same data representation with it.

Another big concern with Captain Simpson's object model was how data transfer occurred between two computers. However, I didn't analyze it in much detail since my intention was to reuse it. By reuse of these C++ routines, features of Ada and the C++ class library interface can be evaluated.

The analysis of the Joystick class proceeded in a manner similar to the analysis of the RS232 port class. One major concern with the Joystick class was the degree of concurrency. In certain modes, the joystick would operate independently of the computer. In addition, the Unix operating system was capable of multiple processes running at the same time in a time sharing mode. The Unix operating system input routines associated with the joystick would be part of a separate process from the one running the flight simulator. Identifying concurrency in the analysis phase made the issue of concurrency easier to handle in the implementation phase because task types in Ada were known.

The methods and attributes needed for each class within their class hierarchy were analyzed. The main concern with analyzing methods and attributes was with their reuse. It seemed that all methods providing for the classes were methods that the object could provide using only the state information contained within it. The rest of the classes in figure 3.1 were analyzed in a similar manner.

3.3 Design

The design of a software system is one of the most important parts of a software development effort. The analysis was done by examining the relationships between the object-oriented model and domain model, by analyzing Ada bindings to C++ and C, and by analyzing the object-oriented model. The design decisions were then made and details added to the model in order to describe and optimize the implementation. The overall idea was to produce a reusable software component that users could use in other applications without modification or with providing parameters. The main goal in designing and implementing each component was the ease of use and reusability (extensibility and maintainability were derived from designing for reusability).

Making the component easy to use for the designer who plans to use the interface of the component is centered totally upon the methods that are offered by the component. When used in this manner, the component can be viewed as a "black box". Information hiding and encapsulation derived from abstraction of the component interface is the overriding principle in making components easy to use.

The main idea to build reusable software component was to look at the software component from the perspective of a potential user of a class. All components were constructed from the standpoint that they may be used in other, possibly unrelated applications. One of the most effective ways of accomplishing this is to look at the component in isolation from the rest of the system being built.

The design of such reusable software components presents the design with a set of characteristics of reusable components. There are several important characteristics of components intended for reuse (18:84). One of the characteristics of reusable components is a component's *interface.* The syntactic interfaces specify compile-time invariants that determine how components fit together, and semantic interfaces specify execution-time invariants that determine what the component computes. Another important characteristic of reusable components is *abstraction,* which was mentioned in the previous chapter as the most powerful tool available to the human. The idea of *function abstraction* is that a function *F* may be specified entirely by an *input - output* relationship. The user of a component based on function abstraction need not know how the function is implemented. Another important abstraction technique is *data abstraction*, in which data as well as function implementation may be hidden from the user. With this abstraction technique and component's interface approach, the hidden data characterizes the current state, which may be transformed by means of the set of internal (hidden) operations. However, it is much more difficult to define how one should go about designing components to exhibit these properties.

Components that incorporate these techniques are usually referred to as *objects* and are said to be object-oriented. The object is a reusable software component having a hidden state and a set of operations or capabilities for transforming the state. OOD has been widely accepted as being a method which is likely to lead to substantially increased software reuse, with abstract objects being perceived as the natural unit of reuse. The class to which such objects belong resemble ADTs in many ways. ADT is a class of objects defined by a set of operations available on them and the abstract properties of these operations.

OOD promotes reuse by means of *interface abstraction* - use of the interface does not require knowledge of the implementation, and *inheritance mechanisms* - inheritance is a mechanism for deriving one abstraction from another, specifying only the difference between the new (derived) and old (parent). This inheritance mechanism establishes a relationship between these abstractions, usually a dependency of the derived class on the parent with the benefit of eliminating the need to recode each new abstraction form scratch. This increases software productivity through abstraction reuse.

The design of the reusable Joystick and RS232 Port components was not considered a main effort since the OOD model was reused. This model focused on reusability with C++. For example, "Unmanaged_RS232_Port" and "Distributed_RS232_Port" were inherited from the abstract class "Port". Given this relationship, the "Joystick" instantiated any type of "Port" and used polymorphic methods to use any type of Port because this polymorphism was provided by C++.

However, redesign was required for adapting to the Ada culture. The first option was to make each Port a component of the Joystick since Ada does not support run-time polymorphism. In addition, two more modifications were performed. The "Managed_RS232_Port" was not inherited from "Unmanaged_RS232_Port" because it acts just the same as the "Unmanaged_RS232_Port". The "Ada_wrapper" for wrapping C++ routines was added to surround the C++ class members and allow them to be gradually replaced later. The modified Joystick and RS232 Port classes OOD model is depicted in figure 3.2.

The second option was to make a pointer to a "Port" class object a component of the "Joystick" class. A pointer to a "Port" class object is now a component of the "Joystick". This relationship makes "Joystick" instantiate any type of "Port" and use polymorphic methods to use any type of Port. This option was taken because it provides a higher abstract view of port objects, and also, Ada 9X provides run-time polymorphism. The design of this

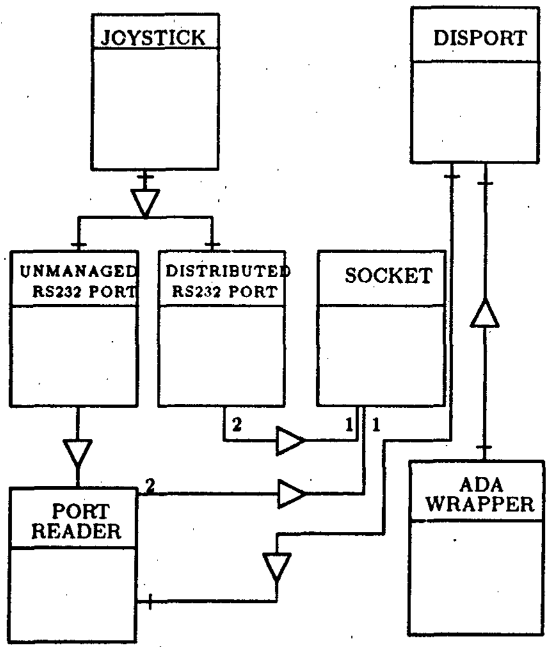

Figure 3.2. Option 1: Design of Joystick and RS232 Port Classes

modification appears in figure 3.3. The detailed design of the Joystick and RS232 port are shown in the Appendix A.

Since one of the main purposes of this thesis was to access existing C++/C routines, the design of the joystick and RS232 Port components were conceptually separated by two subcomponents, one to control an RS232 port for the input on one machine, the other to control an RS232 port for getting data from one machine to the other. The former was intended to have routines written in Ada, and the latter was intended to have routines written in C++. By separating components, this design was able to evaluate the features of Ada interfacing with a C++ classes library through the "Ada_Wrapper" class.

3.3.1 Alternative Methods for Design of a Reusable Component. Several methods for building a well-engineered reusable component were considered. Each alternative achieves the goal of building reusable components by having some effect on OO mechanisms with Ada language features, such as generic units with default formal parameters (both objects

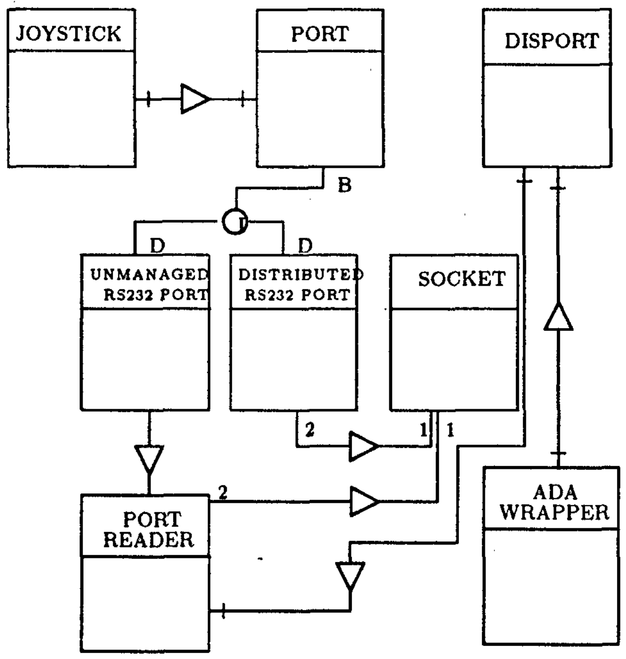

Figure 3.3. Option 2: Design of Joystick and RS232 Port Classes

and subprograms), strong data typing, derived types and subprograms, and subprogram overloading. These alternatives address not only to a limited extent all the fundamental features of the object-oriented paradigm in Ada, but also the conflicting goals that arise in the design of reusable software.

3.3.2 Abstract State Machine (ASM) Approach. The most direct representation of an object is a state encapsulating package (termed abstract state machine approach) exporting a set of operations that can be used to access and update the object state. ASM is a kind of "black-box" approach. The user is provided with a high-level interface to the components. There is no direct access to the data structures themselves. All access is through the operations provided in the interface. Using generic packages, this approach can be extended to emulate a class. The Unmanaged_RS232 port object, for example, can be represented explictly in Ada as an ASM package of form:

```
with port;
```

```
generic
package Unmanaged_RS232_Port is
        ....
        function Close_Port (OB : in object) return boolean;
        procedure Flush_Queue (OB : in object) ;

        ....
package body Unmanaged_RS232_Port is
        type object is record
             port_number : port.port_numbers; – port number of device (minus 1)
             tty_type : tty.termio; – port settings
             port_speed : unsigned_types.unsigned_short_integer; – port speed setting
             .....
        end record;
        ....
```

A generic state-encapsulating package with this interface defines an object template from which multiple structurally identical instances can be generated. However, generic packages are static entities that can only be instantiated at compile time, and thus do not support the concept of dynamically instantiatable objects identified by references, let alone support the accompanying mechanisms of inheritance, polymorphism and dynamic binding.

3.3.3 Task Approach. The second method is to represent objects as tasks. This method can support dynamically instantiatable classes and the notion of concurrency. It can also be used to realize a form of dynamic binding. In this approach the class *Joystick* can be represented by a task type of the form:

```
task type Joystick is
    entry get_coordinates(x_value : in out integer;
                          y_value : in out integer;
                          but1 : in out integer;
                          but2 : in out integer;
                          but3 : in out integer;
                          flag : in out boolean);
    entry Set_Joystick_Mode(new_mode : in joystick_mode);
    .....
    .....
```

One of the main advantages of this approach is that it supports the notion of concurrency and enables objects to be active. It can also be used to realize a form of dynamic binding because Ada permits a task to define several different *'accept'* statements (method bodies) for each entry (method) exported in the interface. By parameterizing the task with a flag indicating which of the *'accept'* statements is to be executed, it is possible for different instances of a single task type to provide alternative implementations for the entry concerned. However, the problem with this method is that classes emulated by task types in Ada can not provide support for inheritance. Another problem is that they can not be library units (5:184-185).

3.3.4 ADT Approach. In addition to the above two approaches, the notion of ADT can be used for representing objects and classes. In this method, an object is defined by a package exporting an ADT. The data structure is declared in the private section in a package specification, thus the user is "stuck" with the data structure provided by the designer. To change it, he must change both the specification where the data structure is defined and the body. This approach differs from the ASM approach in that the interface consists of both the predefined set of operations and the data structure itself, but the state of the data is not captured (14:45). The package exporting the type and associated methods does not itself represent an object but rather variables of the exported data type, and the package defining the ADT corresponds more to a class, therefore, than to an object. The reference semantics and the dynamically instantiated objects are provided by making the exported type an access type rather than a static type. The main advantage of this method is that it provides limited support for two important mechanisms associated with classes/objects, inheritance and polymorphism through the variant and class-wide programming. This can provide a representation of inheritance with respect to the operations associated with a class. The following example shows how the use of this can implement inheritance.

```
Package Port is
   Type Port_Type is (Unmanaged_RS232_Port_Type, Distributed_RS232_Port_Type)
   ...
```

```
    -declare the variant record.
    Type Port_Record is private;
    type Port is access Port_Record;
    subtype Unmanaged_RS232_Port is Port;
    subtype Distributed_RS232_Port is Port;
    .....
    function Unmanaged_Port_Open(Self : Unmanaged_RS232_Port) return boolean;
    function Distributed_Port_Open(Self : Distributed_RS232_Port) return boolean;
    function Port_Open(Self : Port_Type) return boolean;
    .......
    private
    Type Port_Record (Class : Port_Type) is
      record
         Port_Open : Boolean;
         case Class is
         when Unmanaged_RS232_Port_Type =>
           Port_FD : integer;
           Port_Speed : integer;
           ......
         when Distributed_RS232_Port_Type =>
           Data_Socket : Socket;
           Cmd_Socket : Socket;
           ....
         end case;
      end record;
end Port;
```

However, the main shortcoming of the derived type and subtype mechanisms is that they do not permit the set of state variables associated with an abstraction to be extended. Building a new system from pre-existing components is not possible without modification or adaptation to the specific requirements of the new system to fully facilitate reuse. For example, the record type with variant part is used to model Port_Record. The actual structure and processing depends on their Port_Type, which is used as a discriminant. However there are several problems with this approach. For example, wherever code exists to handle Port_Record, case statements must be used to determine the actual subtype of the Port_Record prior to its processing. For example, the following program shows how the actual subtype is determined.

```
   ...
function Port_Open (Self : Port_Type) return boolean is
begin
   case self.Class is
      when Unmanaged_RS232_Port_Type ⇒
         return Unmanaged_Port_Open(Self);
      when Distributed_RS232_Port_Type ⇒
         return Distributed_Port_Open(Self);
      when others ⇒ return false;
   end case;
end
   .....
   .....
```

This variant approach is fragile in maintenance. For example, if a programmer wants to modify the system to support a new Port_Type (e.g., X.21_type), then type Port_Type must be modified, as must type Port_Record, as well as any subprograms that handle Port_Type or Port_Record, even if these operations do not require any additional logic for the X.21_type.

3.3.5 Generic Approach. Another method for representing classes and objects is to use Ada generic units to provide components which are tailorable to user-defined types. It provides flexibility while simplifying use. Types and operations on the types are defined, and the types can then be used to instantiate the generics and the operations will get pulled along. The major advantage of this approach is that it incorporates strong typing and is flexible. However, the user would need to supply a large number of generic parameters. This burden can be alleviated by the judicious use of default parameters. Generics can be exploited in the development of reusable components. Low-level components can be designed as generic packages or subprograms. A set of higher-level parts components can then be built from multiple levels of these generic units. The user provides actual parameters to instantiate the generic components and tailor them to his application. This approach would be used with the ASM or ADT approach. Effective use of generic units for the creation of reusable components requires reconciliation between the complexity of the generic specification and the ease of use of the component.

3.3.6 Using Ada 9X. Ada 9X allows ADTs to have user-defined initialization and finalization and generalizes user-defined equality/inequality for any type. Ada 9X also adds class-wide operations as well as run-time polymorphism within a class of related types as an option for programmers, while retaining Ada's generally static type model.

Ada 9X provides support for the paradigm of object-oriented programming (OOP) through powerful mechanisms for variant and class-wide programming and child library units. For example, the limitations of the Ada 83 ADT approach can be addressed with Ada 9X. With Ada 9X, the programmer can use tagged type extension and subprogram dispatch to simplify the system, handling each variant as a derived type extension, eliminating variant records and case statements. Processing for each kind of Port_Record is localized to a type, and dispatch will insure that the proper operation is called for in each instance. For example, the above Ada 83 code will be translated into Ada 9X code as follows:

```
Package Port is
   type Port is tagged private;
   function Port_Open(Self : Port) return boolean;

   type Unmanaged_RS232_Port is new Port with private;
   function Port_Open(Self : Unmanaged_RS232_Port) return boolean;
   type Distributed_RS232_Port is new Port with private;
   function Port_Open(Self : Distributed_RS232_Port) return boolean;
   .......
   private
   type Port is tagged;
     record
         Port_Open : Boolean;
     end record;
   .......
end Port;
```

Now the variant record type has been replaced with a Port and two types derived from it. Type Unmanaged_RS232_Port extends the Port and has its own Port_Open function. Each derived type inherited the primitive operation of its parent type Port. Each derived type has its own Port_Open procedure, and overrides function Port_Open of the parent function Port_Open. In addition, the new type can contain additional components, and one can define new operations. Instead of the single function Port_Open embodying a case statement as in the Ada 83 solution, the Ada 9X solution distributes the logic for handling Port_Record to each specific Port_Type, without redundancy.

```
function Port_Open(Self : Port) return boolean is
begin
   ......
end
function Port_Open(Self : Unmanaged_RS232_Port) return boolean;
begin
   Port_Open(Port(self));
   ......
end
```

Each body for Port_Open encloses just the code relevant to the type, and delegates additional processing to an ancestor via an explicit type conversion. In Ada 9X, *view conversions* to a tagged class-wide type preserve the tag of the object to permit repeated dispatch within the class determined by the target type.

If a new kind of Port_Type for special purpose must be added, it may be done without disturbance or recompilation of the existing system code as a separate package. The following example program shows how a new variant can be added.

```
with Port;
package Port.New_Port is
   ...
   ...
   type X.21_port is new port.port with private;
   function Port_Open(Self : X.21_Port) return boolean;
```

However, *objects of Port'class* are of unknown, varying size, due to the possibility of extensions. For the Port, the set of values of the class-wide type Port'class is the union of the set of values of Port and all of its derivatives. For this reason, Ada 9X treats them as unconstrained, analogous to unconstrained array types (e.g., string) in Ada 83 (1). For example, when a class-wide type, *Port'class*, appears as the designated type in an access type declaration, the resulting type may designate any object within the class rooted at *Port*. Using such class-wide access types will be a common idiom of OOP in Ada 9X. For example, the following code will be inserted in package *Port*:

```
...
type Port_ptr is access Port'class;
...
function Port_Open(class_Ptr : Port_Ptr) return boolean;
.....
```

Port_ptr is an access type with designated type Port'Class. This allow Port_ptr values to designate objects of type Port, or any derivative of Port. The operation Open_Port is a class-wide operation in that it takes parameters of type Port_ptr, which designates the class-wide type Port'Class. When a primitive operation of a tagged type is called with an operand of the class-wide type, the operation to be executed is selected at run-time based on the type tag of the operand. This run-time selection is called a *dispatching* operation. Dispatching provides Ada programmers with a natural unit form of run-time polymorphism within classes of related (derived) types. This variety of polymorphism is known as "inclusion polymorphism".

Class-wide programming and type extension, in conjunction with generic units, provides many useful facilities. Generic units may be parameterized by user-defined classes, allowing abstractions to be built around such classes. Consider the following example programs in conjunction with *Port*. The generic package Joystick has the following form:

```
with Port;
generic
   type generic_Port(<>) is new port.port;
   ....
   with function Port_Open(self : generic_Port) return boolean;
   .....
package Joystick is
   ...
```

In this example, type *generic_Port* will be matched by any type derived from port.port. This generic package could be instantiated with a specific derivative of port.port and other actual parameters. The notation (<>) specifies that the actual generic_Port type may have any number of discriminants or be a class-wide type. A generic unit may extend a tagged type, adding components and operations. The extended types declared within such generic units inherit all the properties of the original type and process all the new properties defined by the generic units. Such generic units act as "mixin" classes and provide one aspect of multiple inheritance (1).

Now all OO programming mechanisms were provided. The only debate for Ada 9X is multiple inheritance. Ada 9X supports multiple-inheritance via multiple *with/use* clauses, via private extensions and record composition, via the use of generics and formal packages, and via access discriminants (39). However, in Ada 9X, the linguistic multiple inheritance mechanism is not provided because of the potential for distributed overhead caused by multiple inheritance. But this is a minor point: multiple inheritance is a programming style, not a universal tool, and object-oriented practice of the past ten years indicates that the critical benefit of OOP, namely code reuse, is not substantially enhanced by multiple inheritance (33).

3.3.7 Abstract Data Type with Common Class. Using these different methods, it was possible to achieve something of the effect of all the principal object-oriented mechanisms. None of these methods except Ada 9X is acceptable for implementing a general object-oriented language like C++ because they each support only a certain subset of the required properties. For example, if a class is modelled by a task, so as to take advantage of the dynamic binding and concurrency, it is not possible to use inheritance. If the ADT approach with derived types and subtypes is used, it does not permit the set of state variables associated with an abstraction to be extended. Ada 9X is the best solution to implementing a set of reusable components, since it provides support for the paradigm of object-oriented programming (OOP) through powerful mechanisms for variant and class-wide programming and child library units. However Ada 9X was not available at the time of this work.

Thus, this thesis design was implemented with Ada 83. The next solution was the ADT approach. The final alternative was the ADT approach with a common class method. This method eliminates limitations such as type incompatibility with the simple ADT approach. ADT with the common class method was the approach developed and used on the DRAGOON project (5). The main difference between ADT with common class and the simple ADT approach is that the state of object is not represented by a single record but by a linked list of records. Each node in the list stores the state variables added by its ancestors. Thus the state of the objects in this thesis would be represented by a list with the 'node' storing their attributes specific to their objects added to a 'node' storing their attributes belonging to their classes objects. Conceptually, all user-defined application classes except classes for getting data from one machine to the other are descendants of *common_object* and may be assigned to instance variables of this class. Therefore, logically, the first node of every such *state list* corresponds to an instance of the class *common_object*. Figure 3.4 shows the full version the Joystick and RS232 port class hierarchy.

In ADT with a *common_object* class approach, all objects in the component are represented by state lists whose first node is a record of type *Common_Object.State*, referenced by an access variable of type *Common_Object.Object*. Instances of class *Common_Object.Object* are a special case in that their state is represented by a 'uninode' list containing a single record of this type. The Ada record type used to generate this special first node in the state list is defined following a package part of the predefined environment of every Ada library used for implementing components in this thesis.

```
package common_object is
        type state;
        type object is access state;
        type state is record
            offspring_no : natural := 0;
            self : object;
            multiple : object;
```

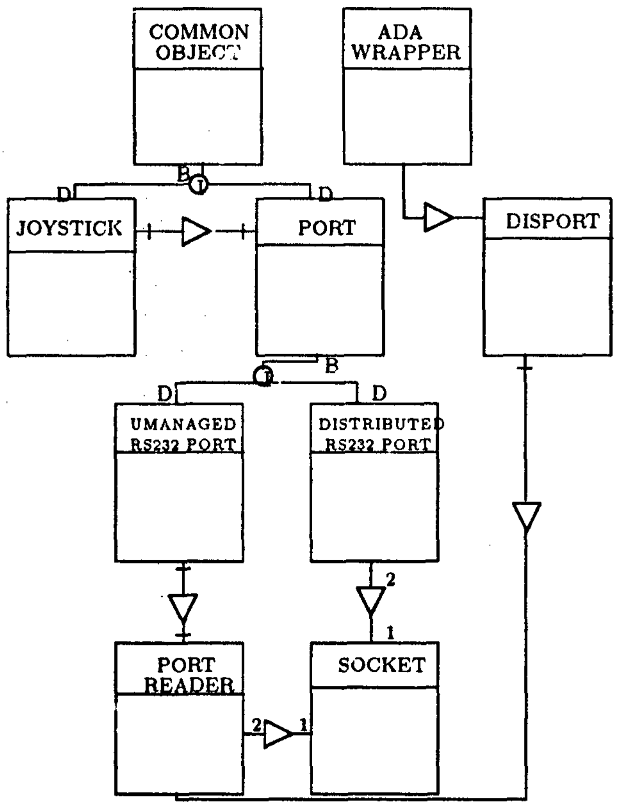

Figure 3.4. Full Version of the Joystick and RS232 Port Class Hierarchy

```
        heir : object;
    end record;
    function CREATE (offspring_no : in natural := 0)
        return object;
end common_object;


package body common_object is
    function CREATE (offspring_no : in natural := 0)
        return object is
        OB : object;
        begin
            OB := new state;
            OB.offspring_no := offspring_no;
            OB.self := OB;
            OB.multiple := null;
```

```
                OB.heir := null;
                return OB;
        end;
end common_object;
```

The state lists corresponding to descendants of *Common_Object.Object* are composed of an *Common_Object.State* record followed by records holding the state variable introduced by each of the classes in the inheritance chain. This is the basic strategy for overcoming the incompatibility of the polymorphism features and Ada's strong typing mechanism. Since the first node of every state list is of the type *Common_Object.State*, all objects in the component are referenced by access values of the same Ada type - *Common_Object.Object.*

The *OFFSPRING_NO* and *MULTIPLE* fields of the *Common_Object.State* are used to handle multiple offspring and multiple inheritance situations, respectively. The field *SELF*, on the other hand, simply points back to the record so that objects may access its own state. The last field *HEIR* is the one that contains the 'links' or references to other nodes to build a linked list for non-trivial objects. In the case of instances of the class *Common_Object*, the *HEIR* field is left containing the value *'null'* since the *'state'* of each objects is represented by an instance of *Common_Cbject.State* alone. However, for objects of descendent classes, this field is used to point to the next node in the list. The *HEIR* field of the *Common_Object.State* is defined to be any type *OBJECT* for convenience, but in fact any access type would have sufficed because it is impossible to predict at the time of its definition what the type of the next node will be. This information is only available when an heir of *Common_Object* is transformed. This is the point at which the Ada typing rules need to be broken so that the nodes representing newly defined heir classes can be 'linked' on to the list corresponding to objects of the parent class. To make this *HEIR* field point to a record with a different type to *Common_Object.Object*, Ada's predefined generic function *UNCHECKED_CONVERSION* must be used to change its apparent type.

The only problem with this approach is that it introduces unnecessary *Common_Object* class and attributes for making each class linked. For example, all classes are inherited from

Common_Object class, and have two additional attribut·s - *Offspring_no* and *Heir* whi·h are not essential to them.

IV. Detailed Design and Implementation

4.1 Introduction

One of the main goals of this thesis was that C++ should be readily translatable into Ada and be able to interface with Ada for building reusable software components. This chapter describes how the object model is translated into Ada in a fairly succinct and natural style.

4.2 Detailed Design and Implementation of the Reusable Port and Unmanaged_RS232_Port Component

The abstract class *PORT* was translated into a package with the following specification and body. The strategy for representing *PORT* objects in Ada was based on the approach of an ADT with a *Common_Object* class. The *PORT* objects are generated by the member function *CREATE*. The *PORT* class package provides the reference semantics and the associated facility for dynamically generating objects by making the exported type an access type rather than a static type:

```
with common_object;
with unchecked_conversion;
package Port is
        type variable;
        type state is access variable;
        type variable is record
             port_open : boolean; -flags if port is open
             offspring_no : natural := 0;
             HEIR : common_object.object;
        end record;
        function common_view_of is new
             unchecked_conversion (source ⇒ port.state,
                                   target ⇒ common_object.object);
        function port_view_of is new
             unchecked_conversion (source ⇒ common_object.object,
                                   target ⇒ port.state);
        function part_of (OB : in common_object.object) return state;
        function create (offspring_no : in natural := 0) return common_object.object;
        function Get_Port_Open (OB : in common_object.object) return boolean;
        function Open_Port (OB : in common_object.object) return boolean;
        procedure Read_From_Port(
            OB : in common_object.object;
            Buffer : in out string;
            num_chars_to_read : in integer;
            count : out integer);
        ..
        .. -other methods
exception : UNDER_FLOW;
end port;

Package body Port is
        .
        . other variables and functions
        function create (offspring_no : in natural := 0)
            return common_object.object is
            com_obj : common_object.object;
            port_obj : state;
        begin
            port_obj := new variable;
            port_obj.offspring_no := offspring_no;
            port_obj.HEIR := null;
            com_obj := common_object.create(1);
            com_obj.HEIR := common_view_of(port_obj);
            return com_obj;
        end;
```

As illustrated in Figure 4.1 and program examples, the state of Port object is represented by a linked list with two nodes, the first node of type *Common_Object.State* and the second of type *Port.variable*, storing port attributes. The job of linking the two nodes together transparent'v is performed by the *CREATE* function using Ada's predefined function *UNCHECKED_CONVERSION.* To enable the *HEIR* field of the first node to point to a record of type *port.variables*, the create function makes references of type *port.state* appear to be of the expected type *Common_Object.Object.* For example, the *port_view_of* function in the example program is an instantiation of the generic function *UNCHECKED_CONVERSION* converting access values of type *port.state* to the type *Common_Object.* Therefore, all objects in the components are of type *Common_Object.State* referenced by an access variable of type *Common_Object.Object.* Not only does this mechanism solve the problem of polymorphism, but it also means that there are no typing obstacles to the incremental introduction of new subclasses, since instances of these are also represented by state lists referenced by the accessed variable of type *Common_Object.Object.* For example, if the *HEIR* field of the first node is 'null', the list represents an instance of class *Common_Object*; if not, then it must correspond to a descendant of *Common_Object,* and therefore can be supplied as a parameter to a method of a descendant class. In order for the 'methods' of the class to manipulate the state variable stored in the corresponding state node, the *Part_Of* function was introduced. It performs the inverse *'UNCHECKED_CONVERSION'* to the *CREATE* function. Given a reference of type *Common_Object.Object*, it returns a reference of the access type defined in its package. The breaking of the type rules is therefore performed transparently in a disciplined manner through the two functions *CREATE* and *PART_OF* within each package.

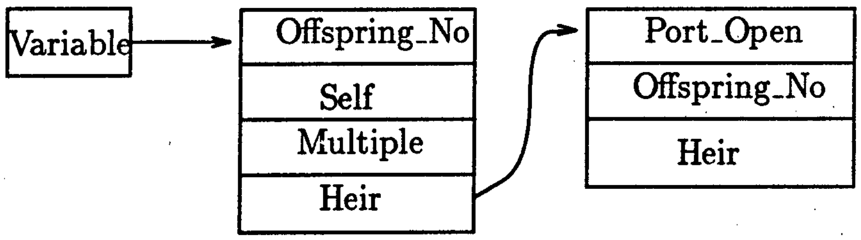

Figure 4.1. List Structure Holding State of Port Objects

4.2.1 Inheritance. As mentioned previously, the inheritance mechanism supports the reuse of an existing ADT as the basis for the definition of the new ADT. This mechanism does not establish any connection between the old one and new one. In this thesis, this inheritance is accomplished through the *linked list,* which has a common object access type. In a linear inheritance hierarchy, the process of adding nodes onto the list is repeated for each new addition to the hierarchy. As illustrated in Figure 4.2, the state of a *Unmanaged_RS232_Port* object is stored as a linked list of three nodes. The Ada package into which

Unmanaged_RS232_Port is translated, however, is completely independent of the record types used to generate the first two nodes in the list. For example, *Unmanaged_RS232_Port* would be translated into a package of the form:

```
with port; with common_object; ......
package Unmanaged_RS232_Port is
        type variable;
        type state is access variable;
        type variable is record
             port_FD : os_files.file_descriptor; - port file descriptor number
             port_type : port.port_comm_type; - terminal, modem or flow control
             ....
             offspring_no : natural := 0;
             HEIR : common_object.object;
        end record;
        ......
        function part_of (OB : in common_object.object) return state;
        function create_of (
             port_num : in port.port_numbers;
             speed : in port.port_speed_spec;
             mode : in port.port_input_mode;
             port_c_type : in port.port_comm_type;
             offspring_no : in natural := 0)
             return common_object.object;
             ...
             ... other functions
             ...
        function Close_Port (OB : in common_object.object) return boolean;
        procedure Flush_Queue (OB : in common_object.object) ;
```

This structure is not affected in any way by that of the package *Port* corresponding to its parent class. The only place in which reference is made to this package is in the implementation of the *Create* and *Part_Of.* The *Create* and *Part_Of* functions would be translated into a function of form:

```
package body Unmanaged_RS232_Port is
   function part_of (OB : in common_object.object) return state is
      begin
         return Unmanaged_RS232_Port_view_of(port.part_of(OB).HEIR);
```

```
    end;
  function create_of (
        port_num : in port.port_numbers;
        ......
        offspring_no : in natural := 0) return common_object.object;
    com_object : common_object.object;
    rs232port : Unmanaged_RS232_Port.state;
    begin
        rs232port := new Unmanaged_RS232_Port.variable;
        rs232port.offspring_no := offspring_no;
        rs232port.HEIR := null;
        ....
        rs232port.port_mode := mode;
        rs232port.port_type := port_c_type;
        com_object := port.create(1);
        port.part_of(com_object).HEIR := common_view_of(rs232port);
        return com_object;
  end Create_of;
  ......
```

The key benefit of this mechanism is in the structure of these two functions. Redefining an abstraction from a pre-existing class is not at all influenced by the implementation of the parent class.

4.2.2 Dynamic Binding. To illustrate the problems involved in implementing this mechanism, consider the class *Unmanaged_RS232_Port.* The most important feature of this class, as far as dynamic binding is concerned, is that it re-implements some of the methods inherited from abstract class *PORT.* Consequently, when one of the redefined methods is invoked through an instance variable of class *PORT*, the particular version of the method which is executed depends on the dynamic type of the instance variable, that is, the type of the object to which it is referring at the time of the call. The problem, therefore, is to decide at run-time which of the Ada subprograms implementing the alternative versions of the method should be executed. Moreover, the incremental development facilities of the OO approach mean that the programmer may define further subclasses at any later stage; the range of different versions that may be invoked does not remain fixed. There must be some

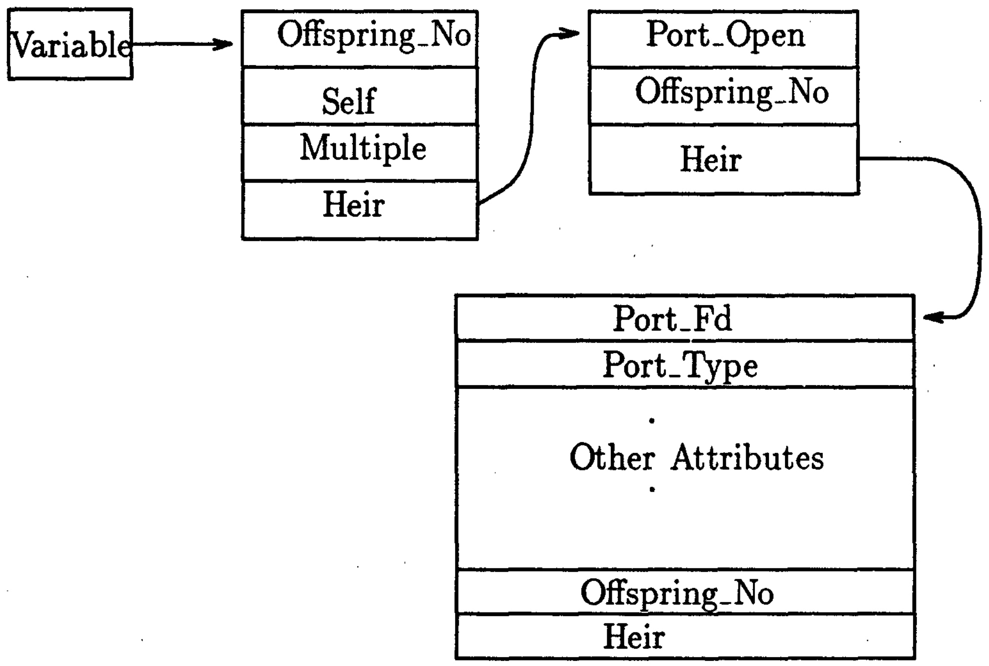

Figure 4.2. List Structure Holding State of Unmanaged_RS232_Port Objects

Ada code in the system that knows about all the different current versions of a method in the system and is able to select the appropriate version at run-time. If this was embedded in the body of the Ada packages into which classes are translated, however, the code would have to be reproduced and recompiled each time a new version of a method was defined in a subclass. The incremental development principle of the OO approach would thus be largely undermined.

Ada's features for defining the bodies of methods in physically separate subunits, however, provides an elegant mechanism for avoiding this problem. It permits the amount of code that has to be updated to cater to the introduction of new method versions to be limited to a single procedure body. None of the subprograms declared in the package specification actually implements the corresponding method directly, however. This job is, in fact, performed by an additional set of methods declared in a package *SELF* contained in the body of the main package. The bodies of the visible subprograms declared in the specification of the main package are contained in *'separate'* units and use the subprograms defined in the inner package *SELF* to implement the original method.

```
Package body Port is
   package self is
     function Open_Port (OB : in common_object.object) return boolean;
     procedure Read_From_Port(
         OB : in common_object.object;
         Buffer : in out string;
         num_chars_to_read : in integer;
         count : out integer);
     .....
end self;

   package body self is
     function Open_Port (OB : in common_object.object) return boolean is
         begin
           if port.part_of(OB) = null then
           raise UNDER_FLOW;
           else
           return false;
           end if;
         end;
     ....
     procedure Read_From_Port(
         OB : in common_object.object;
         Buffer : in out string;
         num_chars_to_read : in integer;
         count : out integer);
         begin
           raise UNDER_FLOW;
         end;
   end self;
   function Open_Port (OB : in common_object.object)
         return boolean is separate;
   procedure Read_From_Port(
      OB : in common_object.object;
      Buffer : in out string;
      num_chars_to_read : in integer;
      count : out integer) is separate;
   .
   . other functions
```

```
end port;
```

The subprograms declared in the inner package *SELF* contain the Ada image of the code in the body of the corresponding methods. For example, the *Read_From_Port* method declared in the inner package *SELF* contains the Ada image of the code in the body of the *PORT.*

Since *PORT* is an HEIR only of *COMMON_OBJECT* and does not inherit any user-defined method, when it is first implemented into Ada, there is only one version of each of its methods known to the system. Until the subclass of *PORT* is added to the library, therefore, the subprograms declared in the specification of the corresponding Ada package are essentially redundant. The only action they perform is to call the corresponding method contained in *SELF.* The body of the exported *Open_Port* subprogram, for example, has a separate body of the form:

```
separate (Port)
function Open_Port (OB : in common_object.object) return boolean is
   begin
     return self.Open_Port(OB);
   end;
```

At this stage, this subprogram makes no useful contribution to the implementation of the method. This occurs when the programmer defines new versions of the method in subclasses. The subclass of *PORT* that does this is the class *Unmanaged_RS232_Port* which redefines the *Open_Port.* When *Unmanaged_RS232_Port* is implemented into Ada, the separate body of the subprogram *PORT.Open_Port* is replaced by the following:

```
with Unmanaged_RS232_Port;
separate (Port)
function Open_Port (OB : in common_object.object) return boolean is
   begin
   if port.part_of(OB) = null then
     return self.Open_Port(OB);
   else
```

```
    return Unmanaged_RS232_Port.Open_Port(OB);
  end;
```

When invoked, this function analyzes the form of the state list *OB* to see whether the state list represents an instance oı *PORT* or its subclass *Unmanaged_RS232_Port*. If it represents the latter, it invokes the *Unmanaged_RS232_Port* version, otherwise it invokes the method in the package *SELF*. Essentially, therefore, this function forms a kind of 'shell' around the true method implementations in order to select, at run-time, the appropriate one for execution.

The great advantage of this approach is that all the modification and recompilation needed to cater to the new version is limited to the 'separate' subprograms of the methods concerned. This advantage come from the separation of the method selection subprogram; that is, it is replaced as a subunit of the main package. Not even the body of the package, let alone the Ada code for clients of the class, needs to be recompiled when methods redefining subclasses are added to the system.

This technique is fine for distinguishing between the different versions of a method that may be introduced in a linear inheritance chain, that is, when each class has only one parent and one child class. However, if a parent has more than one child class, before using a *Part_Of* function to convert its type, it is essential to determine to which of the child classes the next node in the list actually corresponds. Another field is needed in the nodes of the state lists to indicate in which of the branches of the inheritance tree the class represented by the subsequent node lies. The *Offspring_no* is the purpose of indicating which of the child classes is the next node in the list. Together, the *Heir* and *Offspring_no* fields of state nodes provide all the information needed by selection shells to determine which version of a method to execute in response to a call. Suppose, for example, that the *Distributed_RS232_Port* was translated into the package, which also redefined the *Open_Port*. Now the *Offspring_no* field would be assigned natural number 1. In order to determine the appropriate implementation when *Open_Port* is invoked through an instance variable of class *Distributed_RS232_Port*, the body of the *Open_Port* function (selection shell) would be replaced by the following form:

```
with Unmanaged_RS232_Port;
with Distributed_RS232_Port;
separate (Port)
function Open_Port (OB : in common_object.object) return boolean is
   begin
   if port.part_of(OB) = null then
     return self.Open_Port(OB);
   else
   case port.part_of(OB).offspring_no is
   when 0 =>
     return Unmanaged_RS232_Port.Open_Port(OB);
   when 1 =>
     return Distributed_RS232_Port.Open_Port(OB);
   when others =>
     raise UNDER_FLOW;
   end;
```

4.2.3 Clientship. The method of implementing classes in this thesis makes the translation of client code very straightforward. All instance variables, of whatever class type, are translated into Ada *access* variables of type *CommonObject.Object* since the first node of all state lists is of type *CommonObject.Object.* The translation of method invocations employs the same principle used in the simple ADT approach. That is, the Ada access variable corresponding to the called object is supplied as the first parameter of the subprogram implementing the method. Thus a method 'Read_From_Port' call of *Unmanaged_RS232_Port* would be translated into the following subprogram invocations:

```
Port.Read_From_Port
(part_of(OB).instantiated_RS232,part_of(OB).JOY_noise_buffer, JOY_DATA_SIZE,count);
                    ...
                    ...
```

where *instantiated_RS232* is a instance variable and *JOY_noise_buffer*, *JOY_DATA_SIZE* and *count* are variables for method invocation. The separated Port.Read_From_Port method would select the method *Unmanaged_RS232_Port* class at run-time. Now, *instantiated_RS232* is an access variable of type *Common_Object.Object.* Similarly, the generation of objects by invocation of *CREATE* method is simply translated as follows:

```
instantiated_RS232 : common_object.object;
                   .
                   .
instantiated_RS232 :=
                   unmanaged_RS232_port.create_of(port_num, port_speed, port_mode, port_type)
                   .
                   .
```

4.3 Detailed Design and Implementation of Reusable a Joystick Component

As mentioned in the previous chapter, *Joystick* class has a degree of concurrency. The joystick would in certain modes operate independently of the computer. The task is the unit of concurrency in Ada. The implementation of active objects, with their concurrent execution threads, must clearly be based on the use of tasks. Because a task is defined in terms of actions rather than statements or instructions, even the execution of a single program, such as a procedure which prints "Hello" on a terminal, can be viewed as a single, implicit task whose thread of execution runs in parallel with the rest of the system. In Ada, tasks allow the programmer to decompose a problem into several independent threads of control. These techniques enable a programmer to model different activities in the real world simultaneously. For example, an avionics system has altitude, radar, joystick, and a graphics display, each of which is continually monitored for valid reading. Additionally, the graphics display is updated periodically to reflect position, altitude, velocity, and terrain. Each of these subsystems can be modeled by a task. These tasks are independent activities.

The one problem in using a task to represent the thread of an object is in integrating it with the state list representation of objects used so far. However, this can be overcome because Ada permits tasks generated from task types to be identified by access variables that can be included in the appropriate record structure. In addition to the fields storing the state variables of the objects, the state node of active objects has an additional field holding a reference to a task. The active class *Joystick* can be translated into a package with a specification of the form:

```
package Joystick is
type variable;
type state is access variable;
type thread_form;
type thread_ref is access thread_form;
type variable is record
      RS_232port : common_object.object;
      JOY_mode : character;
      JOY_out_mode : character;
      JOY_buffer : Buffer_type;
      JOY_noise_buffer : buffer_type;
      ......
      ......
      joy_x, joy_y : integer;
      joy_1, joy_2, joy_3 : integer;
      offspring_no : natural := 0;
      HEIR : common_object.object;
      thread : thread_ref;
end record;
task type thread_form is
      entry get_coordinates( OB : in common_object.object;
                             x_value : in out integer;
                             y_value : in out integer;
                             but1 : in out integer;
                             but2 : in out integer;
                             but3 : in out integer;
                             flag : in out boolean);
end thread_form;
function Joystick_view_of is new
      unchecked_conversion( source ⇒ common_object.object,
                            target ⇒ state);
procedure get_coordinates (
                             OB : in common_object.object;
                             x_value : in out integer;
                             y_value : in out integer;
                             but1 : in out integer;
                             but2 : in out integer;
```

```
                                        but3 : in out integer;
                                        flag : in out boolean);
    .....
    .....
```

The procedure *get_coordinates* corresponds to the task entry *get_coordinates* method used to activate the thread of active classes. Invocation of the *get_coordinates* method by a client of an active object is thus translated into the invocation of the *get_coordinates* procedure whose body is of the form:

```
package body joystick is
         .....
         .....
         procedure get_coordinates( OB : in common_object.object;
                                    x_value : in out integer;
                                    y_value : in out integer;
                                    but1 : in out integer;
                                    but2 : in out integer;
                                    but3 : in out integer;
                                    flag : in out boolean) is
         begin
                                    joystick.part_of(OB).thread := new thread_form;
                                    joystick.part_of(OB).thread.get_coordinates
                                    (OB,x_value,y_value,but1,but2,but3,flag);
         end get_coordinates;
         .....
         .....
end joystick;
```

The first action performed by this procedure is to instantiate the task type *thread_form* and assign its access value to the *thread* field of the state node associated with *Joystick*. The procedure then calls the *get_coordinates* method of the task to give it the reference to the state list so that the thread may manipulate the variable of the objects. The calling of this entry also serves to unblock the tasks so that it may begin execution of the code corresponding to the body of the thread. The *thread_form* task type therefore has the following body:

```
package body joystick is
```

```
          .....
          .....
task body thread_form is
          begin
          loop
          accept get_coordinates( OB : in common_object.object;
                                   x_value : in out integer;
                                   y_value : in out integer;
                                   but1 : in out integer;
                                   but2 : in out integer;
                                   but3 : in out integer;
                                   flag : in out boolean) is
          joystick.read_joystick(OB,x_value,y_value,but1,but2,but3,flag);
          end get_coordinates;
          end loop;
end thread_form;
          .....
          .....
end joystick;
```

Once the *get_coordinates* procedure has generated an instance of task type *thread_form* and provided it with a reference to the object's state list by calling its *get_coordinates* entry, the task will execute concurrently with other threads and method invocations, as required. Moreover, as it is included in the state node of the corresponding objects, it is intimately associated with the corresponding object state for the duration of the program. For example, consider simplified versions of the flight simulator task units "request_task_type" which may trigger a long input operation task "get_coordinates". Each "request_task_type" task calls task "get_coordinates" to determine the current position of a moving object, based on the X,Y-coordinates of the corresponding joystick. These tasks communicate by sending each other not only synchronization information, but data as well. The message passing concept in Ada is called the *rendezvous*. After the *rendezvous* is complete, the two tasks continue independently.

4.4 Detailed Design and Implementation of the Ada Wrapper

The *Adawrapper* class was intended as a wrapper class in which C++ class members of data and function were interfaced with Ada code. Basically, two approaches were taken; one through the *C-linkage* (e.g., extern "C" { }, which says that everything within the scope of the brace-surrounded block is compiled by a C compiler), and the other through the type-safe linkage and name encoding techniques. This section describes the problems involved in generating names for overloaded functions in C++ and in linking to Ada and C++ programs. It also discusses how problems referred to in this thesis were solved.

The type-safe linkage and name encoding technique (C++-linkage) discussed in this thesis was based on the 2.0 release of C++. C++, like a Ada, allows overloading of function names; two functions may have the same name provided their argument types are different, while C does not provide function name overloading. C has a simple naming convention for external symbols, which includes global variable and function names. The C compiler just prepends an underscore character '_' to external symbols. This simple scheme clearly isn't sufficient to cope with overload functions. However, in C++, every function name is encoded by appending its signature.

The C++ function name encoding scheme was originally designed primarily to allow the function and class names to be reliably extracted from encoded class member names. The basic approach is to append a function's signature to the function name. According to the AT&T C++ *Language System Selected Readings* (4), the function name encoding scheme under C++ version 2.0 is defined as shown in table 4.4.

A global function name is encoded by appending __F followed by the signature so that, for example, Read_Packet(int, char, float) becomes *Read_Packet__Ficf*. Within Ada, this function should be called through the encoded name 'Read_Packet__Ficf'. Thus when the port was being set up for "raw" input allowing the port to receive inputs present in the read queue (regardless of whether the tty device is done sending a full packet or not), the tty.set_tty_state Ada function was used . This function was provided by the VADS 'verdixlib'

Types	Encoded	Modifiers	Encoded
void	v	unsigned	U
char	c	const	C
short	s	volatile	V
int	i	signed	S
long	l	pointer *	P
float	f	reference	&R
double	d	array	[10]A10_
long double	r	function ()	F
		ptr to member	S::*MIS

Table 4.1. The C++ Type-safe Linkage and Name Encoding Techniques

library. However, the function didn't set up the port correctly. Additional parameters were set up and blocked the terminal. For the safety of the terminal set, a C++ function was written to set the port up for "raw" rather than Ada function. Then the C++ function was called within Ada. To bind with C++, the parallel data types between Ada and C++ were created. Creation of parallel data types was the same as C, which meant that C and C++ have basically the same data representations. Then the C++ function name encoding scheme was used to access that C++ function within Ada.

```
with system;
    ...
Package Unmanaged_RS232_Port is
    ...
procedure c_port_port( port_FD_num : in system.address,
                       ttyport : in system.address;
                       port_speed : in system.address;
                       P_mode : in system.address);
                       ...
pragma INTERFACE (C, c_port_open);
pragma INTERFACE_NAME (c_port_open, & "c_port_open__FPiPcN2");

Package Body Unmanaged_RS232_Port is
                       port_FD_num : integer;
                       ttyport : string(1..11);
                       P_mode : integer;
                       ...
```

```
                    ttyport(1..8) := "/dev/tty";
                    ttyport(11) := ascii.nul;
                    P_mode := port.port_input_mode'pos(part_of(OB).port_mode);
                    c_port_open(port_FD_num'address,ttyport'address,
                    part_of(OB).port_speed'address,P_mode'address);
                    ...
end Unmanaged_RS232_Port
```

Corresponding C++ program

```
c_port_port(int *FD, char path [], int *port_speed, int *P_mode)
{
                    ...
}
```

Another way of accessing C++ global functions within Ada was to use the C-linkage instead of the C++-linkage. The *extern "C"* statement means that everything within the scope of the brace-surrounded block is compiled by a C compiler. With this approach, the function was accesses through *pragma INTERFACE* within Ada. All procedures to access C++ functions within Ada are basically the same as that of the C++-linkage except for the encoded function name.

```
pragma INTERFACE (C, c_port_open);
pragma INTERFACE_NAME (c_port_open, C_SUBP_PREFIX & "c_port_open");
```

Corresponding C++ program

```
extern "C" c_port_port(int *FD, char path [], int *port_speed, int *P_mode)
{
    ...
}
```

Stroustrup suggested the linkage from C++ to another language as follows: "I conjecture that in most cases linkage from C++ to another language is best done simply by using a common and fairly simple convention such as 'C-linkage' plus some standard library routines and/or rules for argument passing, format conversion, etc., to avoid building knowledge of non-standard calling conventions into C++ compilers" (4:6-9). As he suggested, the use of 'C-linkage' instead of C++-linkage made interfacing with Ada simpler for unique name global C++ functions.

However, there are several problems with 'C-linkage' for overloading functions and class members. The first is a safety problem with function overloading. 'C-linkage' basically can not overload functions, since two functions with the same function name and different signatures can cause serious side effects. The second, and more serious, problem with 'C-linkage' was related to a class and its members. A linkage specification for classes applied to only non-member functions and objects declared within it. There was no way of using 'C-linkage' for C++ classes and their members, which means that every linkage specification for classes and their members should use C++ naming encoding techniques.

According to the C++ name encoding technique, names of classes was encoded as the length of the name followed by the name itself to avoid terminators. For example, the member function of joystick class, Set_Y_Normalize(int&), becomes Set_Y_Normalize__8JoystickFRi. The procedure of binding with this class member function was basically the same as that of C++ global functions. The details within Ada are as follows:

```
procedure Set_Y_Normalize (N1 : in integer);

pragma INTERFACE (C,Set_Y_Normalize);
pragma INTERFACE_NAME(Set_Y_Normalize,"Set_Y_Normalize__8JoystickFRi");
```

The main problem with this approach is that the instances of C++ classes are not exported from C++ to Ada. That is, Ada could not instantiate the C++ class from within Ada because class definition in C++ does not cause any memory to be allocated. Memory is allocated for a class with the definition of each class object.

At first, an intermediate C routine which transfers C++ class structure to Ada was tried. The problem was the same as with Ada's case. Neither Ada nor C can export C++ class data structures to create instances of C++ classes. One possible way of exploiting a class library from Ada was to use pointers to class members and itself, which was the first approach taken. It seemed possible because we were able to create parallel data structures corresponding to pointers of class data members and declare subprograms corresponding to class function members according to the C++ naming encoding rules. In C++, a pointer to an object of a class points to the first byte of that region of memory. The C++ compiler turns a call of a member function into an "ordinary" function call with an extra argument; that extra argument is a pointer to the object for which the member function is called (4:5-2). For example, a simple class Joystick:

```
class Joystick {
                int x,y,but1,but2,but3;
                void read_joystick(int x,int y,int but1, int but2, int but3);
                .....
};
```

A call of the member function Joystick::read_joystick:

```
    Joystick *ptrjoystick; -pointer to Joystick class
    ptrjoystick⇒read_joystick(x,y,but1,but2,but3);
```

is transformed by the compiler into an "ordinary function call":

```
    read_joystick__F8Joystick(ptrjoystick,x,y,but1,but2,but3);
```

From the above ordinary function call, Ada may be able to access individual C++ class members through another intermediate C++ global function which just creates an instance of a class. However, we were not able to exploit class members within Ada from the C++ class library. Later, we found out there was no pointer to a class member under C++ compiler version 2.0

Another way of exploiting the C++ classes library was to build a C++ main function which instantiates the classes and invokes the member functions. Then Ada can access the C++ main function directly through Ada *pragma INTERFACE.* Within C++ main, object attributes (parameters) are passed to the Ada routine by calling Ada subprograms. For example, the Ada main program calls the C++ main program just like calling the C++

global functions. And the C++ main program passes object data members as parameters by invoking the Ada subprogram within the Ada main program.

```
procedure joyadatest is
procedure joyt;
pragma interface (C, joyt);
pragma interface_name (joyt,"joy__Fv");
procedure getdata( x_val : in integer;
                   y_val : in integer;
                   but_1 : in integer;
                   but_2 : in integer;
                   but_3 : in integer;
                   flag : in integer);
pragma EXTERNAL (C, getdata);
pragma EXTERNAL_NAME (getdata, C_SUBP_PREFIX & "givedata__FiN51");
procedure getdata(x_val : in integer;
                   y_val : in integer;
                   but_1 : in integer
                   but_2 : in integer;
                   but_3 : in integer;
                   flag : in integer) is
begin
                   -perform something
end getdata;
begin
                   joyt;
end;

C++ program
extern "C"
{
                   int x_val = 0;
                   other declarations
}
extern givedata (int, int. int, int, int, int);
Joystick *ptr;
joy()
{
                   Joystick jstick(Port::port_four);
                   ptr = &jstick;
                   ptr⇒Set_Y_Normalize(TRUE);
                   ptr⇒Set_Y_Resolution(10);
```

```
            while ((!(but_1)))
            {
              flag = jstick.Read_Joystick(&x_val, &y_val, &but_1, &but_2, &but_3);
              givedata(x_val,y_val,but_1,but_2,but_3,flag);
            }
}
```

For the Adawrapper class, we first tried to replace *DISTPORT* class with an Ada routine, because its only function is to create a *PORT READER* class object and call the read method. However, we couldn't find the way of directly exploiting a class library within Ada. The *Adawrapper* was needed to access C++ class *DISTPORT*. Actually the *DISTPORT* was the main program that instantiates a class *'Port Reader'*, then runs on a remote machine. The *DISTPORT* was invoked by passing the parameters, which are the command line arguments in *argc* and *argv*.

```
with system; use system;
with language; use language;
with command_line; use command_line;
procedure distportada is
      procedure distport( argc : in system.address;
                          argv : in system.address);
      pragma interface (C, distport);
      pragma interface_name (distport,C_SUBP_PREFIX & "main");
      begin
                          distport(argc'address,argv'address);
end;
```

Appendix B includes some programs to help in the understanding of the example program code explained in this chapter. All programs included in Appendix B were implemented for developing the reusable joystick component for a flight simulator application domain.

V. Summary and Conclusions

This chapter summarizes the research discussed in this thesis and also presents conclusions.

5.1 Summary

The objective of this thesis was to develop a set of reusable software components for investigating and for demonstrating Ada's applicability as an implementation language for a reusable graphical software component.

A set of components, Reusable Joystick and RS232_Port, were developed for a flight simulator. The OO approach was applied to the implementation of these components using the Ada programming language associated with C++ components.

The development of this thesis started with an analysis of the flight simulator domain. The main purpose of the analysis phase was to identify potential reusable software components. This came from a domain analysis that led to the identification of common objects, operations, and structures. A class was a set of objects that share a common structure and common behavior. Each object class was a candidate for a reusable component for the domain. This thesis identified low level inputs (Joystick and RS232 port classes) as a reusable components implementation.

The next step was to analyze the joystick and the RS232 port classes and their relationship among their class members in more detail. As a part of this, Ada binding to C++/C was analyzed. For a complete Ada binding to a C++/C library and existing codes, the following steps are necessary.

- Create parallel data types.
- Interface to external symbols (routine and data).
- Link to external libraries.

Whenever access to routines or variables declared in the C++/C language is required, any Ada variables used in conjunction with the subroutines or variables are compatible with the C++/C data representation. Once these parallel types have been established, the next step is to gain access to external routines and data provided in the interface target language. This is accomplished in a two stage procedure: first, equivalent Ada subprogram specifications are written, and second, the linkage to the external routine and data is declared. The first step is a simple mapping of external routine and data names and parameters into an Ada subprogram specification. The second step is accomplished through use of the *pragma INTERFACE* and *pragma INTERFACE_NAME*. Linking the symbols built in Ada with the external C++/C symbols is done by the Ada linker.

Then design decisions were made and details were added to the model to describe and optimize the implementation. The main goal in designing and implementing each component was ease of use and reusability (extensibility and maintainability were derived from designing for reusability). The design of such reusable software components resulted in a design which incorporates an *interface* and an *implementation*, resulting in the design of an *abstraction*. Components that incorporate such characteristics are usually referred to as *objects* and are said to be object-oriented. OOD of reusable Joystick and RS232 Port components was used. This model focused on reusability in C++. Redesign was required for adapting to the Ada culture. The "Ada_wrapper" for wrapping C++ routines was added to surround the C++ class members and allow them to be gradually replaced later.

Several methods for building a well-engineered reusable component were considered. Each alternative achieves this goal of building reusable components by demonstrating some effect of Object-Oriented (OO) mechanisms through Ada language features.

The most direct representation of an object is a state encapsulating package exporting a set of operations which can be used to access and update the object state. With this Abstract State Machine (ASM) approach, the user is provided with a high-level interface to the components. All access is through the operations provided in the interface. A generic state-encapsulating package with this interface defines an object template from which multiple structurally identical instances can be generated. However, generic packages are static entities that can only be instantiated at compile time and thus do not support the concept of dynamically instantiatable objects identified by references, let alone support for the accompanying mechanisms of inheritance, polymorphism and dynamic binding.

The second method is to represent objects as tasks. This method can support dynamically instantiatable classes and the notion of concurrency. It can also be used to realize a form of dynamic binding. However, the problem with this method is that classes emulated by task types in Ada cannot provide support for inheritance. Another problem is that they cannot be library units.

In addition to the above two approaches, the notion of Abstract Data Type (ADT) can be used for representing objects and classes. In this method, an object is defined by a package exporting an ADT. This approach differs from the ASM approach in that the interface consists of both the predefined set of operations and the data structure itself, but the state of the data is not captured. The package exporting the type and associated methods does not itself represent an object but rather variables of the exported data type, and the package defining the ADT corresponds more to a class, therefore, than to an object. The reference semantics and the dynamically instantiated objects are provided by making the exported type an access type rather than a static type. The main advantage of this method is that it provides limited support for two important mechanisms associated with classes/objects, inheritance and polymorphism. However, it introduces limitations such as type incompatibility by deriving from the parent types, narrowing their applicability by subtyping, recompilation of the original abstraction by breaking the original abstraction, and complicated generic parameters.

Another method for representing classes and objects is to use Ada generic units to provide components which are tailorable to user-defined types. It provides flexibility while simplifying use. Types and operations on the types are defined, and the types can then be used to instantiate the generics, and the operations will get pulled along. The major advantage of this approach is that it incorporates strong typing and is flexible. However, the user would need to supply a large number of generic parameters.

Ada 9X was considered as the best solution since it provides support for the paradigm of object-oriented programming (OOP) through powerful mechanisms for variant and class-wide programming and child library units. All limitations with the Ada 83 ADT approach can be addressed with Ada 9X. With Ada 9X, the programmer can use tagged type extension and subprogram dispatch to simplify the system. Tagged types offer Ada programmers a mechanism for single inheritance. For a type T, the class-wide type T'Class was introduced. The set of values of T'Class is the union of the sets of values of T and all of its derivatives. The type tag, associated with each value of a tagged class-wide type, is the basis for adding run-time polymorphism in Ada 9X. However, an Ada 9X compiler was not available.

The last alternative considered was an ADT with a common class approach. The main difference between this approach and the simple ADT approach is that the state of the object is not represented by a single record but by a linked list of records. Each node in the list stores the state variables added by its ancestors. In this approach, all objects in the component are represented by state lists whose first node is a record of type *Common_Object.State*, referenced by an access variable of type *Common_Object.Object*. These state lists corresponding to descendants of *Common_Object.Object* are composed of a *Common_Object.State* record followed by records holding the state variable introduced by each of the classes in the inheritance chain. Since the first node of every state list is of the type *Common_Object.State*, all objects in the component are referenced by access values of the same Ada type *"Common_Object.Object"*. This was the basic strategy for overcoming the incompatibility of the polymorphism, inheritance mechanism and dynamic binding without introducing type incompatibility and recompilation while preserving Ada's strong typing mechanism. The only problem with this approach is that it introduces unnecessary *Common_Object* classes and attributes for making each class link. For example, all classes are inherited from the *Common_Object* class, and have two additional attributes - *Offspring_no* and *Heir* which are not essential to them.

This ADT with common class approach was selected for implementation. The inheritance and polymorphism mechanisms were accomplished through the *linked list* which has a common object access type. All objects in the system are of type *Common_Object.State* referenced by an access variable of type *Common_Object.Object.* Not only does this mechanism solve the problem of polymorphism, but it also means that there are no typing obstacles to the incremental introduction of new subclasses, since instances of these are also represented by state lists referenced by accessed variables of type *Common_Object.Object.*

For example, if the *HEIR* field of the first node is 'null', the list represents an instance of class *Common_Object*; if not, then it corresponds to a descendant of *Common_Object* and therefore can be supplied as a parameter to a method of a descendant class.

In a linear inheritance hierarchy, the process of adding nodes to the list is repeated for each new addition to the hierarchy. As illustrated in the previous chapter, the state of an *Unmanaged_RS232_Port* object is stored as a linked list of three nodes. The Ada package into which *Unmanaged_RS232_Port* is translated, however, is completely independent of the record types used to generate the first two nodes in the list. The *Unmanaged_RS232_Port* structure is not affected in any way by that of the package *Port* corresponding to its parent class. The only place in which reference is made to this package is in the implementation of the *Create* and *Part_Of*. The key benefit of this mechanism is in the structure of these two functions.

Ada's features for defining the bodies of methods in physically separate subunits provides an elegant mechanism of dynamic binding. It permits the amount of code that has to be updated to cater to the introduction of new method versions to be limited to a single procedure body. None of the subprograms declared in the package specification actually implements the corresponding method directly, however. This job is, in fact, performed by an additional set of methods declared in a package *SELF* contained in the body of the main package. The bodies of the visible subprograms declared in the specification of the main package are contained in *'separate'* units and use the subprograms defined in inner package *SELF* to implement the original method. The great advantage of this approach is that all the modification and recompilation needed to cater to the new version is limited to the 'separate' subprograms of the methods concerned.

The *Adawrapper* class was intended as a wrapper class, in which C++ class members of data and function were interfaced with Ada code. Basically, two approaches were taken; one through the *C-linkage* (e.g., extern "C" { } which says that everything within the scope of the brace-surrounded block is compiled by a C compiler), and the other through the type-safe linkage and name encoding techniques – C++-Linkage.

One way of accessing C++ global functions within Ada was to use the C-linkage instead of the C++-linkage. The *extern "C"* statement says that everything within the scope of the brace-surrounded block is compiled by a C compiler. With this approach, the function was accessed through *pragma INTERFACE* within Ada. All procedures to access C++ functions within Ada are basically the same as that of the C++-linkage except for encoded functions. The use of 'C-linkage' made Ada interfacing with C++ simple for uniquely named global C++ functions.

However, there are several problems with 'C-linkage' for overloading functions and class members. The first is a safety problem with function overloading. 'C-linkage' basically cannot overload functions, since two functions with the same function name and different signatures can cause serious side effects. The second, and more serious, problem with 'C-linkage' was related to a class and its members. A linkage specification for classes applied to only non-member functions and objects declared within it. There was no way of using 'C-linkage' for C++ classes and their members, which means that every linkage specification for classes and their members should use C++ naming encoding techniques.

Another way of accessing C++ functions was to use "C++ linkage". A global function name is encoded by appending _F followed by the signature so that, for example, Read_Packet(int, char, float) becomes *Read_Packet__Ficf* since, within Ada, this function should be called the encoded name 'Read_Packet__Ficf'. Names of classes are encoded as the length of the name followed by the name itself to avoid terminators. For example, the member function of the joystick class, Joystick::Set_Y_Normalize(int&) becomes Set_Y_Normalize__8JoystickFRi. The procedure of binding with this class member function was basically the same as that of C++ global functions.

The main problem with using C++ linkage was that the instances of C++ classes are not exported from C++ to Ada. That is, Ada could not instantiate the C++ class from within Ada. Neither Ada nor C export C++ class data structures to create instances of C++ classes. One way of exploiting C++ class libraries was to build a C++ main function which instantiated the classes; then Ada accessed the C++ main function directly through Ada *pragma INTERFACE.*

5.2 Conclusions

One of the objectives of this thesis was to build a set of reusable flight simulator components in Ada using an OO approach. I wanted to end up with joystick and RS232 port components that were reusable, maintainable and extensible. Object-oriented techniques with Ada promised to provide a way to achieve these goals. I believe that these goals have been accomplished.

The use of Ada, however, couldn't itself guarantee that a component would be readily reusable. There were a number of important design guidelines that can greatly enhance the reusability of components. These guidelines relate to the design and structure of reusable components which were provided by the OO approach.

However, there were limitations with Ada 83 such as type incompatibility and recompilation, or introduction of attributes which are not essential to an object. Ada 9X addressed these limitations.

Another objective of this thesis was to build a set of reusable Ada software components associated with C++ routines. I wanted to end up with the advanced Ada language features that could access the C++ library. However, it was not possible to export members of classes and objects from class libraries within Ada. It was possible to access C++ global functions and user-defined data types, but not user-defined classes. However, one feasible way was to use "ordinary' function calls with an extra argument; that extra argument is a pointer to the object for which the class member function is called.

It is currently practical to use Ada for graphics applications if graphics libraries are written in C rather than C++. However, it is not practical to use Ada for C++ class graphics libraries.

5.3 *Recommendations*

There are many different areas of the design and implementation techniques to build reusable graphics software components that could be extended and improved. This thesis work does not provide the best solution to the question *"how to develop reusable software components in Ada which are associated with a C++ class library ?"*, that is, it is not suitable for a "cookbook" approach. This thesis has addressed several ways of building a reusable graphics software component in Ada associated with C++ routines using an OO approach.

As Ada 9X translators becomes available, this work can be implemented much more cleanly and directly in Ada 9X. To best address reusability with respect to interfacing with C++, Ada 9X implementations should add interface facilities which enable the Ada 9X translator to choose a storage layout for objects of the named types and user-defined classes in C++ to match the representation that the C++ compiler uses. In addition, it must be possible for an Ada object of any of the types, including storage layout for objects of user-defined classes in C++, to be passed as a parameter to a C++ function with the corresponding formal parameter.

Appendix A. Joystick and RS232Port Design

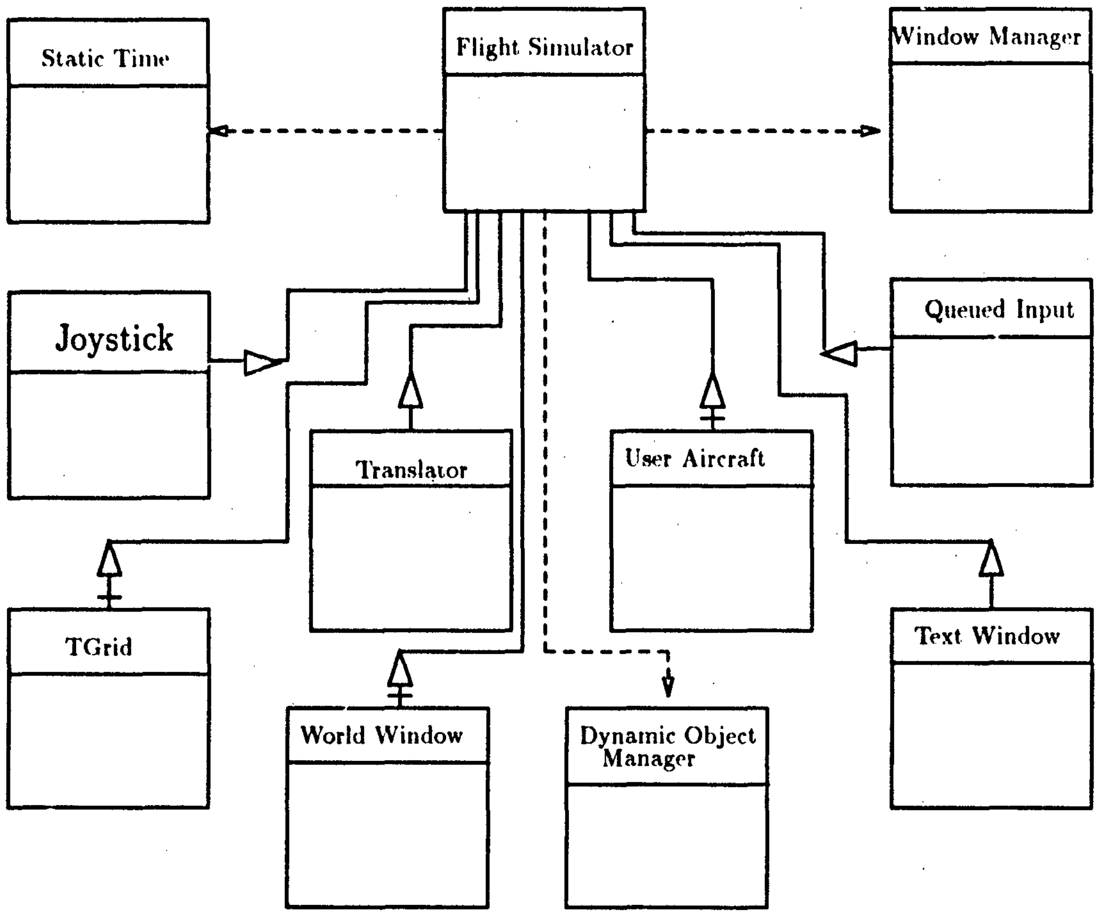

Figure A.1. Flight Simulator Composition

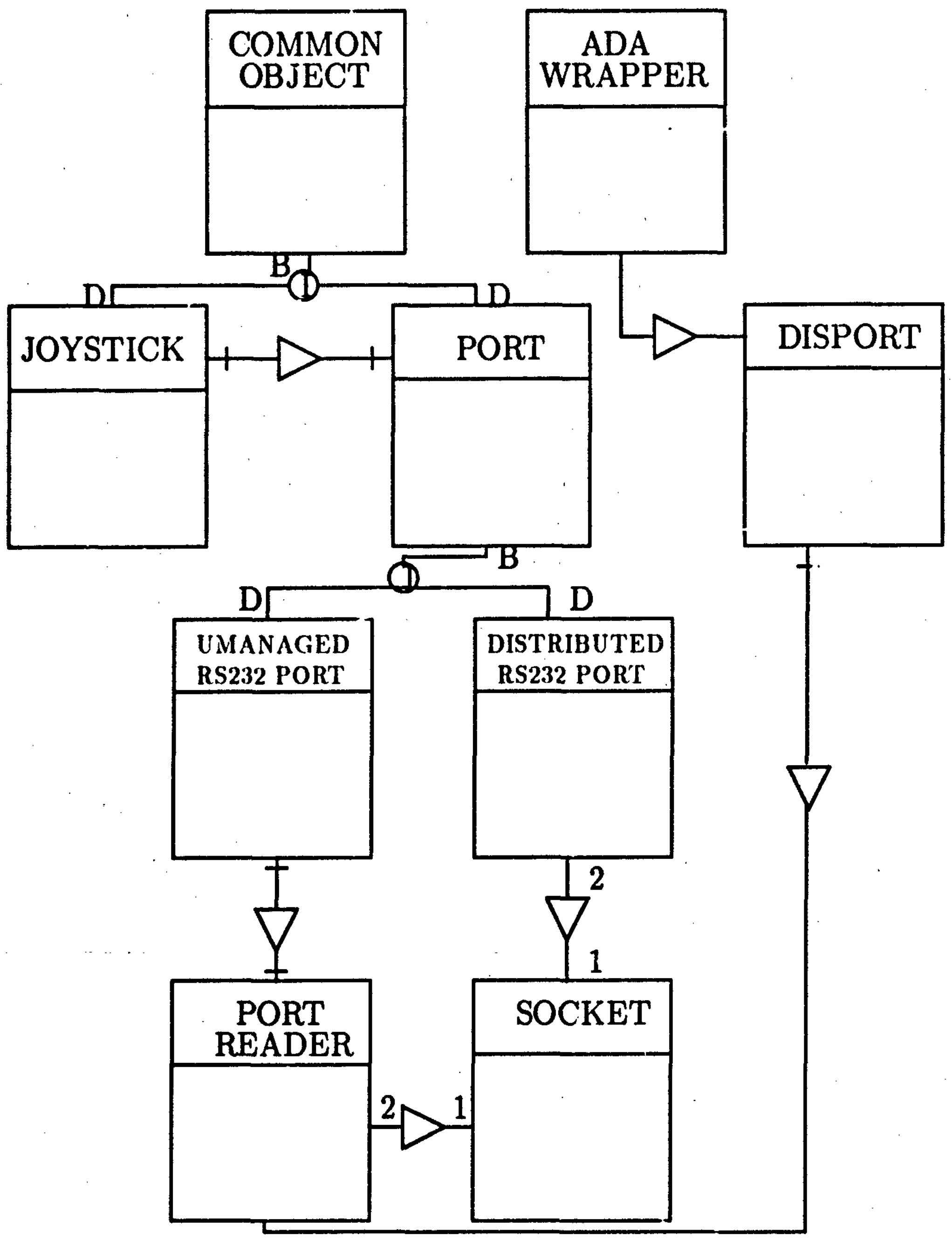

Figure A.2. Joystick and RS232 Port Class Hierarchy

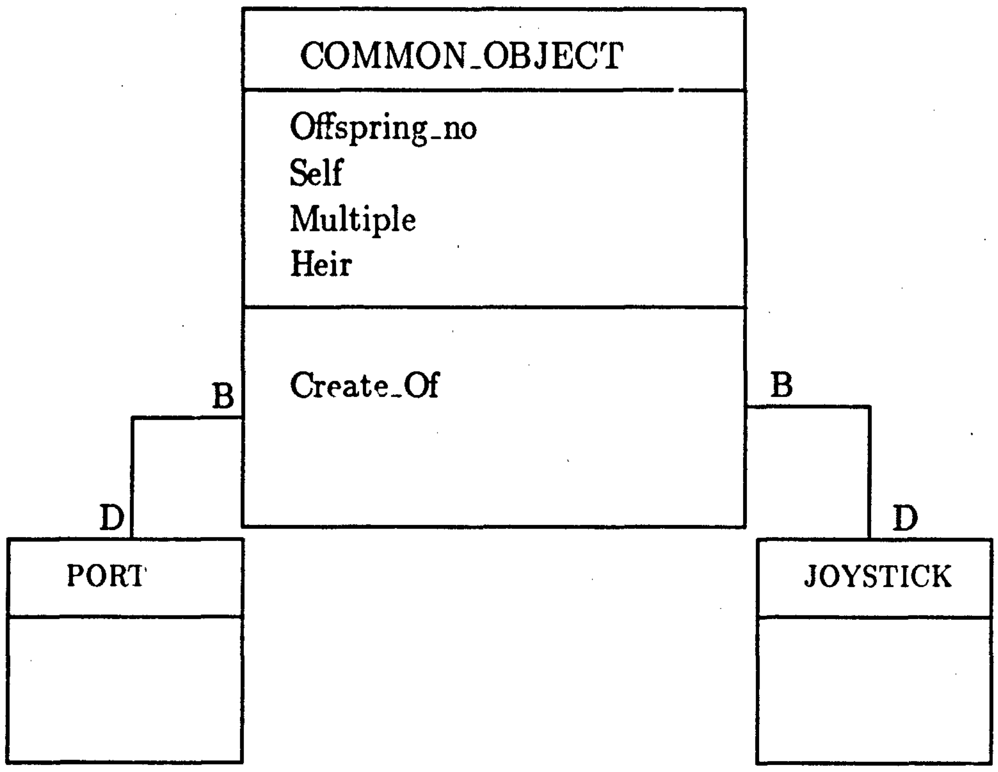

Figure A.3. Common_Object Class

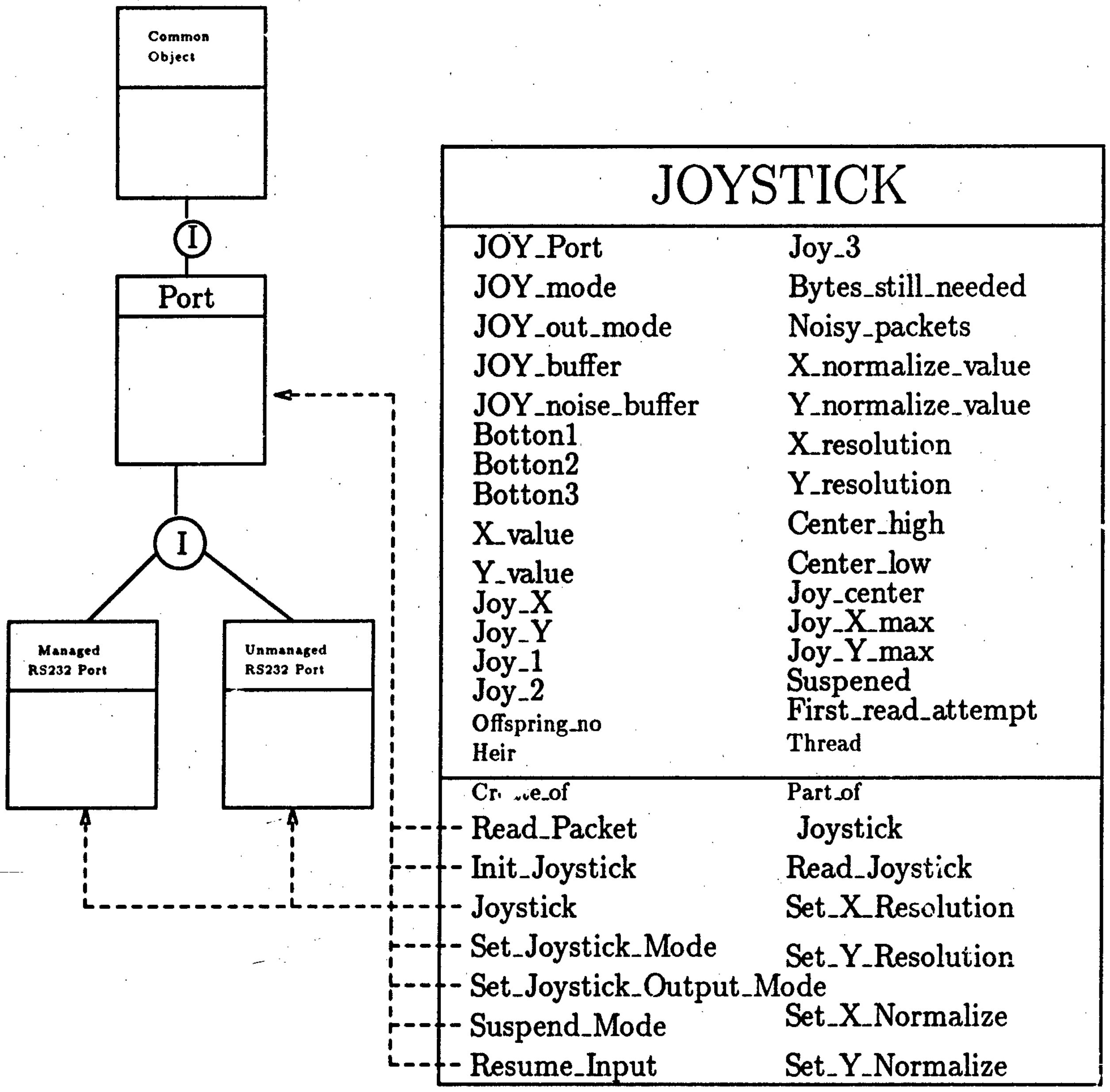

Figure A.4. Joystick Class

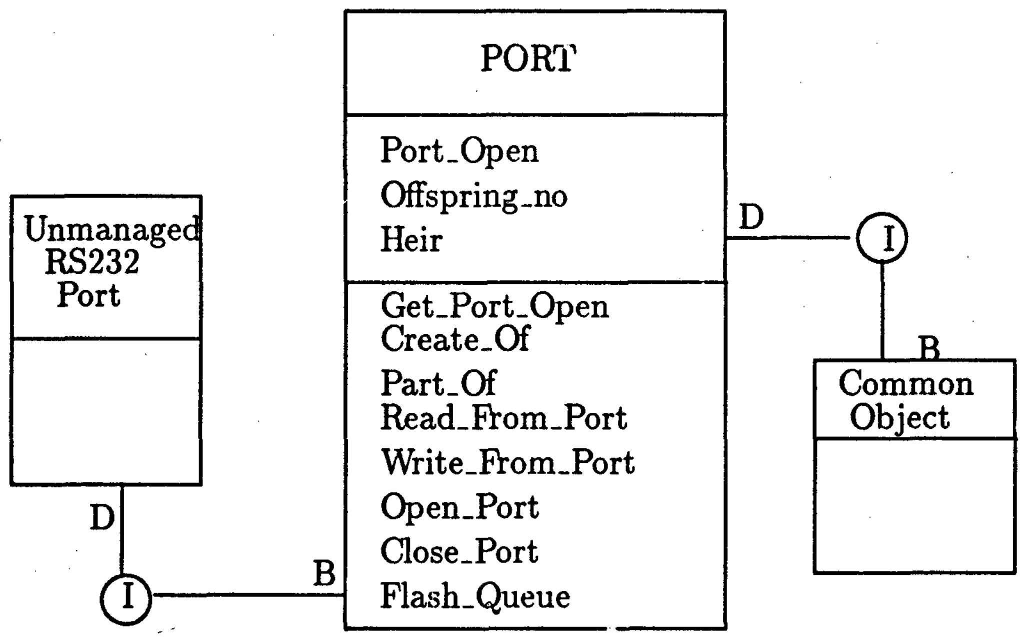

Figure A.5. Port Class

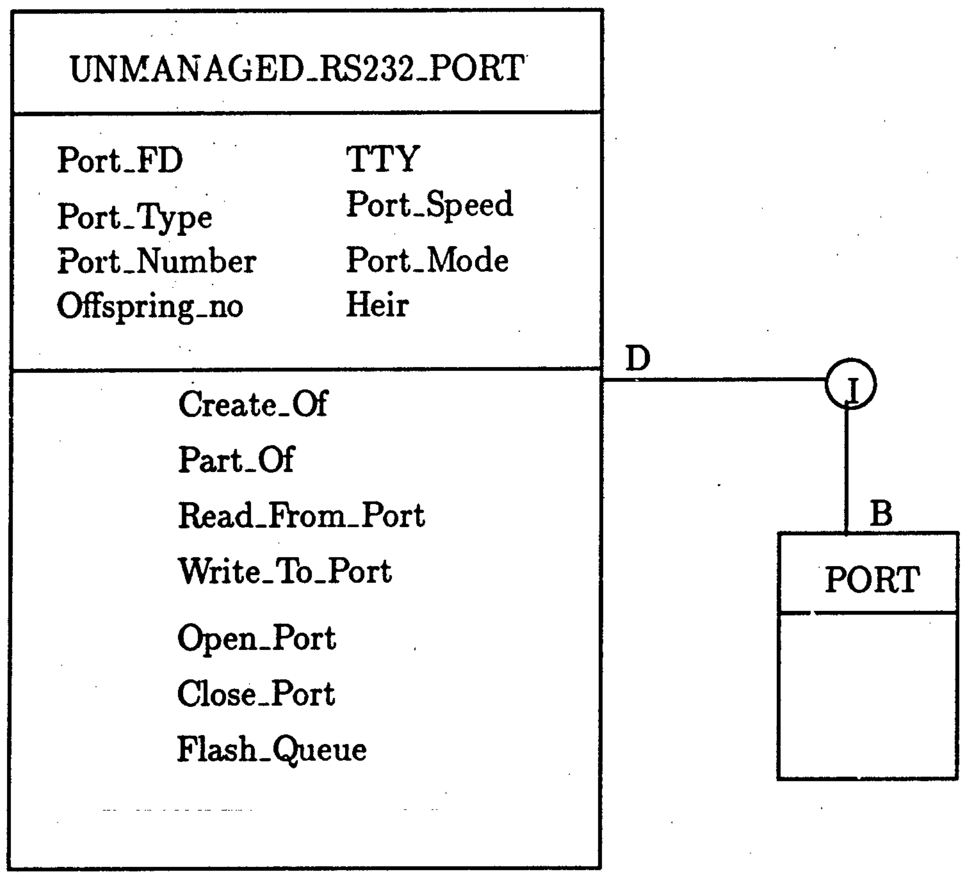

Figure A.6. Unmanaged_RS232_Port Class

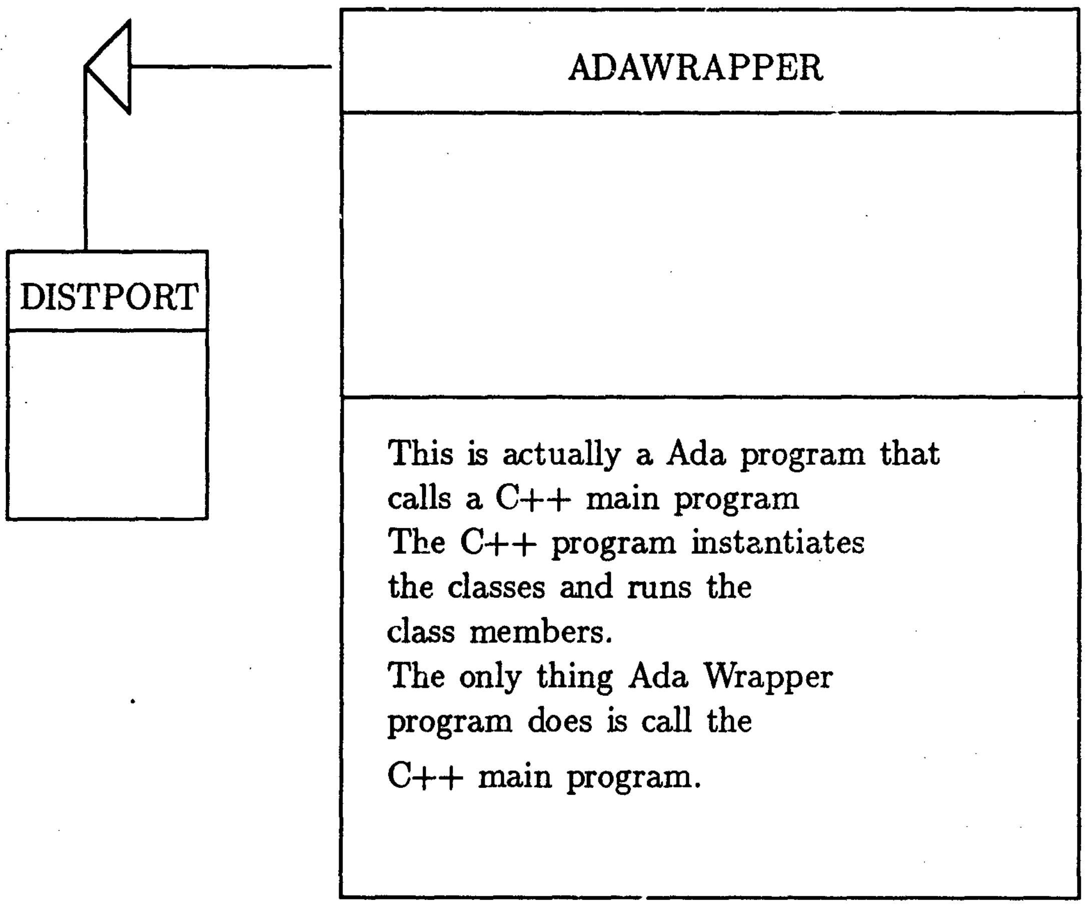

Figure A.7. Adawrapper

Appendix B. Example Source Program List

```
\\************************************************************************
\\File name  : C_Port_Open.cc                                            *
\\Purpose    : It reset the variables in the termio structure to represent*
\\             the way in which you wish the port to behave.  ' .on open the*
\\             port.                                                      *
\\************************************************************************
extern "C" {
#include <sys/termio.h>
int ioctl (int, int, char*);
}
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
extern "C" void c_port_open(int *FD, char path[],int *port_speed, int *P_mode)
{
   int    port_FD;
   struct termio tty;
    if ((port_FD = open(path, O_RDWR | O_NDELAY)) == -1)
      {
       fprintf(stderr,"Cannot open port.\n");
       fprintf(stderr,"Most likely cause is that permissions file.\n");
      }
    else
      {
      ioctl(port_FD, TCGETA, (char *) &tty);
      /* Now reset the variables in the termio structure to represent the way
         in which you wish the port to behave */
/*
 *  Set the port up for:
 *   Hang up on last close
 *   eight bits
 *   local line
 *   enable receiver
 *   enable signals
 *   user specified baud rate
 */
      if (*P_mode == 0) /* input mode is raw */
{
         fprintf(stderr,"Raw mode\n");
        /* These flags set up the port for "raw" input.  This allows us to
          grab whatever input is present in the read queue regardless of
          whether the device is done sending : full packet or not.*/
tty.c_cflag = HUPCL | CS8 | CLOCAL | CREAD | *port_speed;
tty.c_lflag = 0;
tty.c_iflag = IGNBRK;
tty.c_oflag = 0;
tty.c_cc[VMIN] = 0;
tty.c_cc[VTIME] = 0;
}
      else
{
        /* This sets up the port for "canonical" input.  The read queue will not
        make a packet available to the read routine until a <cr><lf> is
        received. */
        fprintf(stderr,"Port_mode is canonical\n");
tty.c_cflag = HUPCL | CS8 | CREAD | *port_speed | CLOCAL;
tty.c_lflag = ISIG | ICANON;
        tty.c_iflag = IGNBRK;
        }
      /* Now set up the port using the TCSETA call to ioctl.  This resets the
         port and flushes the output queue */
    ioctl(port_FD, TCSETAF, (char *) &tty);
    }
  *FD = port_FD;
  }

--****************************************************************************
-- CLASS : Port
-- PURPOSE: This is an abstract class used to implement RS232 ports.
-- FILENAME: port.a
-- METHODS: create -- Constructor, common_view_of, port_view_of,part_of,
--          common_view_of, port_view_of.
--          Read, Write_To_Port, Open_Port, Close_Port, Flush_Queue
--          are all virtual functions to be implemented by lower level
--          classes.
--          Get_Port_Open - returns the only attribute of Port.
-- DESIGN DECISIONS: This standardizes the protocol of the Port classes.
--*****************************************************************************
with common_object;
with unchecked_conversion;
package Port is

  type port_numbers is (port_one, port_two, port_three, port_four,
                port_five, port_six, port_seven, port_eight);
  type port_speed_spec is (b19200, b9600, b1200, b300);
  type port_input_mode is (raw, canonical);
  type port_comm_type  is (terminal, modem, flow_control);
  type variable;
  type state is access variable;
  type variable is record
       port_open : boolean;   --flags if port is open
       offspring_no : natural := 0;
       heir : common_object.object;
  end record;

  ---This function is an instantiation of the generic
  ---function UNCHECKED_CONVERSION converting the type Common_Object
  ---to access values of type.
  function common_view_of is new
           unchecked_conversion (source => port.state,
                                 target => common_object.object);

  ---This function is an instantiation of the generic
  ---function UNCHECKED_CONVERSION converting access values of type
  ---port.state  to the type Common_Object.
  function port_view_of is new
           unchecked_conversion (source => common_object.object,
                                 target => port.state);

  ---It performs the inverse UNCHECKED_CONVERSION to the CREATE
  ---function. Given a reference of type Common_Object.Object,
  ---it returns a reference of the access type defined its package.
  function part_of (OB : in common_object.object) return state;

  ---The purpose of this function is linking the two nodes together
  ---transparently. It enable the heir field of the first node to
  ---point to a record of the other node, and make them reference
  --- of type Common_Object.Object.
  function Create (offspring_no : in natural := 0)
                return common_object.object;

  ---This function returns the only attribute of Port.
  function  Get_Port_Open (OB : in common_object.object)
                         return boolean;
  function Open_Port (OB : in common_object.object) return boolean;

  function Close_Port (OB : in common_object.object)  return boolean;

  procedure Flush_Queue (OB : in common_object.object) ;

  procedure Delete_Unmanaged_RS232_Port (OB : in common_object.object) ;

  procedure Read_From_Port(OB                 : in common_object.object;
                           Buffer             : in out string;
                           num_chars_to_read  : in     integer;
                           count              : out    integer);

  procedure Write_To_Port(OB                  : in common_object.object;
                          buffer              : in out character;
                          num_chars_to_write  : in     integer);

  UNDER_FLOW : exception;
end port;



package body Port is
  package self is
    function Open_Port (OB : in common_object.object)
                  return boolean;
    function Close_Port (OB : in common_object.object)
                  return boolean;
    procedure Flush_Queue (OB : in common_object.object);
    procedure Delete_Unmanaged_RS232_Port (OB : in common_object.object);
    procedure Read_From_Port(OB                 : in common_object.object;
                             Buffer             : in out string;
                             num_chars_to_read  : in     integer;
                             count              : out    integer);
    procedure Write_To_Port(OB                  : in      common_object.object;
                            buffer              : in out character;
                            num_chars_to_write  : in     integer);              end self;
  package body self is
    function Open_Port (OB : in common_object.object)
                  return boolean is
       begin
        if port.part_of(OB) = null then
           raise UNDER_FLOW;
        else
           return false;
        end if;
       end;
    function Close_Port (OB : in common_object.object)
                  return boolean is
       begin
         if port.part_of(OB) = null then
             raise UNDER_FLOW;
         else
             return false;
         end if;
       end;
    procedure Flush_Queue (OB : in common_object.object) is
       begin
         raise UNDER_FLOW;
       end;
    procedure Delete_Unmanaged_RS232_Port (OB : in common_object.object) is
       begin
         raise UNDER_FLOW;
       end;
    procedure Read_From_P                        : in common_object.object;
                                fer              : in out string;
                             num_chars_to_read : in     integer;
                             count             : out    integer) is
       begin
         raise UNDER_FLOW;
       end;
    procedure Write_To_Port(OB                   : in common_object.object;
                            buffer               : in out character;
                            num_chars_to_write : in     integer) is
       begin
         raise UNDER_FLOW;
       end;
  end self;
 function part_of (OB : in common_object.object) return state is
    begin
       return port_view_of(OB.heir);
    end;
 function create (offspring_no : in natural := 0)
                  return common_object.object is
    com_obj  : common_object.object;
    port_obj : state;
    begin
        port_obj := new variable;
        port_obj.offspring_no := offspring_no;
        port_obj.heir := null;
        com_obj := common_object.create(1);
        com_obj.heir := common_view_of(port_obj);
        return com_obj;
     end;
 function Get_Port_Open (OB : in common_object.object)
                         return boolean is
   tem_OB : state;
   begin
     tem_OB := new variable;
     tem_OB := part_of(OB);
     return tem_CB.port_open;
   end Get_Port_Open;
  function Open_Port (OB : in common_object.object)
                      return boolean is separate;
  function Close_Port (OB : in common_object.object)
                       return boolean is separate;
  procedure Flush_Queue (OB : in common_object.object)
                         is separate;
  procedure Delete_Unmanaged_RS232_Port (OB : in common_object.object)
                                        is separate;
  procedure Read_From_Port(OB                  : in common_object.object;
                           Buffer              : in out string;
                           num_chars_to_read : in     integer;
                           count               : out    integer)
                           is separate;
  procedure Write_To_Port(OB                   : in common_object.object;
                          buffer               : in out character;
                          num_chars_to_write : in     integer)
                          is separate;
end Port;

*********************************************************************
*** File neme : Separate_Read_From_Port.a***************************
*** Purpose   : This Read_From_Port method was separate from main *
***             Port package to implement rethod selection sell at*
***             run-time. This reduces recompilation. *************
*********************************************************************
with Unmanaged_RS232_Port;
```

```
separate (Port)
  procedure Read_From_Port
            (OB                 : in common_object.object;
             Buffer             : in out string;
             num_chars_to_read  : in     integer;
             count              : out    integer) is
  begin
    if port.part_of(OB) [illegible] then
        self.Read_From_[illegible]OB,Buffer,num_chars_to_read,count);
  else
        Unmanaged_RS2[illegible]ort.Read_From_Port
                    (OB,Buffer,num_chars_to_read,count);
     end if;
  end;


--**************************************************************************
-- CLASS:  Unmanaged RS232 Port
-- PURPOSE: This allows the user to access an RS232 port.  This does no
--   checking of current port usage.
--
-- INHERITED FROM: Port
-- FILENAME: uRS232port_s.a
-- METHODS: Create_of, part_of, Common_View_of, Unmanaged_RS232_Port_view,
--          Read_From_Port
--          Write_To_Port
--          Open_Port
--          Close_Port
--          Flush_Queue - clears the queue for the port
--
-- DESIGN DECISIONS: Different types of ports were created to give the user
--   a range of different types of ports and use them all in the same way.
--**************************************************************************
with port;     use port;
with system;
with language; use language;
with tty;
with os_files;
with common_object;
with unchecked_conversion;

package Unmanaged_RS232_Port is

type variable;
type state is access variable;

type variable is record
   port_FD     : os_files.file_descriptor;  -- port file descriptor number
   port_type   : port.port_comm_type;       -- terminal, modem or flow control
   port_number : port.port_numbers;         -- port number of device (minus 1)
   tty_type    : tty.termio;                -- port settings
   port_mode   : port.port_input_mode;          -- port mode
   port_speed  : integer;
   offspring_no : natural := 0;
   heir         : common_object.object;
end record;

  ---This function is an instantiation of the generic
  ---function UNCHECKED_CONVERSION converting the type Common_Object
  ---to access values of type.
  function common_view_of is new
          unchecked_conversion (source => state,
                                target => common_object.object);
  ---This function is an instantiation of the generic
  ---function UNCHECKED_CONVERSION converting access values of type
  ---port.state  to the type Common_Object.
  function Unmanaged_RS232_Port_view_of is new
           unchecked_conversion (source => common_object.object,
                                 target => state);

  ---It performs the inverse UNCHECKED_CONVERSION to the CREATE
  ---function. Given a reference of type Common_Object.Object,
  ---it returns a reference of the access type defined its package.
  function part_of (OB : in common_object.object) return state;

  ---The purpose of this function is linking the two nodes togethe
  ---transparently. It enable the heir field of the first node to
  ---point to a record of the other node, and make them reference
  --- of type Common_Object.Object.
  function create_of (
                 port_num     : in port.port_numbers;
   speed        : in port.port_speed_spec;
   mode         : in port.port_input_mode;
   port_c_type  : in port.port_comm_type;
                 offspring_no : in natural := 0)

                 return common_object.object;

---The purpose of function is to open the system ports.
---It performs reseting the variables in the termio structure to represent
---the way in which you wish the port to behave through C++ routine called
---''c_port_open".
function Open_Port (OB : in common_object.object) return boolean;
procedure  c_port_open(port_FD_num : in system.address;
                       ttyport     : in system.address;
                       port_speed  : in system.address;
                       P_mode      : in system.address);

---The purpose of function is to close the system ports.
function Close_Port (OB : in common_object.object)  return boolean;

---The purpose of function is clear the queue for the port.
procedure Flush_Queue (OB : in common_object.object) ;

---The purpose of function is delete the object.
procedure Delete_Unmanaged_RS232_Port (OB : in common_object.object) ;

---The purpose of function is read a data available on the system port.
procedure Read_From_Port(OB                 : in common_object.object;
                         Buffer             : in out string;
                         num_chars_to_read  : in     integer;
                         count              : out    integer);

---The purpose of function is set a specific port mode.
procedure Write_To_Port(OB                  : in common_object.object;
                        buffer              : in out character;
                        num_chars_to_write  : in     integer);

pragma INTERFACE( C, c_port_open);
pragma INTERFACE_NAME( c_port_open, C_SUBP_PREFIX & "c_port_open");

end Unmanaged_RS232_Port;



with text_io;        use text_io;
with tty;
with os_files;        use os_files;
with unsigned_types;  use unsigned_types;
with ioctl;
with unix;
package body Unmanaged_RS232_Port is
   package INT_IO is new integer_io(integer);
   use INT_IO;
   package port_FD_IO is new integer_io(os_files.file_descriptor);
   use port_FD_IO;
   package port_numbers_io is new enumeration_io(port.port_numbers);
   use port_numbers_io;
   package port_comm_type_io is new enumeration_io(port.port_comm_type);
   use port_comm_type_io;
   package port_input_mode_io is new enumeration_io(port.port_input_mode);
   use port_input_mode_io;
--*****************************************************************************
-- METHOD: Part_of
-- PURPOSE: Given OB, it returns a reference of the access type defined
--          its package
--*****************************************************************************
   function part_of (OB : in common_object.object) return state is
      begin
         return Unmanaged_RS232_Port_view_of(port.part_of(OB).heir);
      end;
--*****************************************************************************
-- METHOD: Create_of
-- PURPOSE: This sets up the port according to the user's specifications.
--*****************************************************************************
  function create_of (
                  port_num     : in port.port_numbers;
   speed        : in port.port_speed_spec;
   mode         : in port.port_input_mode;
   port_c_type  : in port.port_comm_type;
                  offspring_no : in natural := 0)
                  return common_object.object is
  com_object : common_object.object;
  rs232port  : Unmanaged_RS232_Port.state;
  begin
    rs232port := new Unmanaged_RS232_Port.variable;
    rs232port.offspring_no := offspring_no;
    rs232port.heir := null;
    rs232port.port_mode := mode;
    rs232port.port_type := port_c_type;
    -- set port number to one less than actual in order to make array access
    -- easier.  Set port type for hardware handshaking that must be done
    rs232port.port_number := port_num;
    -- Load the speed variable according to the user's request and using
    -- the symbolic constants found in termio.h.  Be advised that the
    -- dip switches on the bottom of the device must be set for the speed you
    -- want since this can't be set by the computer. This merely sets how fast
    -- the RS232 PORT will receive stuff.
    case  speed is
      when port.b19200 =>
           rs232port.port_speed := integer(tty.B19200);
      when port.b9600 =>
           rs232port.port_speed := integer(tty.B9600);
      when port.b1200 =>
           rs232port.port_speed := integer(tty.B1200);
      when port.b300  =>
           rs232port.port_speed := integer(tty.B300);
    end case;
    com_object := port.create(1);
    port.part_of(com_object).heir := common_view_of(rs232port);
    port.part_of(com_object).port_open := FALSE;
    return com_object;
  end Create_of;
--*************************************************************************
-- METHOD: Open_Port
-- PURPOSE: Given the settings created when the object was instantiated, this
--    does the UNIX calls to set up and open the RS232 port.
--*************************************************************************
function Open_Port (OB : in common_object.object) return boolean is
  port_FD_num  : integer;
  ttyport      : string(1..11);
  P_mode       : integer;
  T            : boolean := True;
  begin  ---main open_port
   if  not Port.get_Port_Open(OB) then
     ttyport(1..8) := "/dev/tty";
     if part_of(OB).port_type = port.terminal then   -- port type is terminal
       ttyport(9) := 'd';
       text_io.put_line("ttyport(9) = d");
     elsif part_of(OB).port_type = port.modem then -- port type is modem
       ttyport(9) := 'm';
       text_io.put_line("ttyport(9) = m");
     elsif part_of(OB).port_type = flow_control then  -- port type is flow control
       ttyport(9) := 'f';
       text_io.put_line("ttyport(9) = f");
     end if;
     ttyport(10) := character'val(port.port_numbers'pos
                    (part_of(OB).port_number) + 1 + character'pos('0'));
     ttyport(11) := ascii.nul;
     --Call the C program that calls the system calls;
     P_mode := port.port_input_mode'pos(part_of(OB).port_mode);
     c_port_open(port_FD_num'address,ttyport'address,
                 part_of(OB).port_speed'address,P_mode'address);
     part_of(OB).port_FD :=
         os_files.file_descriptor'(os_files.file_descriptor(port_FD_num));
     if port_FD_num /= -1 then
        port.part_of(OB).port_open := TRUE;
     end if;
  end if;
  return port.Get_Port_Open(OB);
 end Open_Port;
--*************************************************************************
-- METHOD: Close_Port
-- PURPOSE: Closes the port (if it was open) and releases the UNIX fd.
--*************************************************************************
function Close_Port (OB : in common_object.object)  return boolean is
  begin
    if port.Get_Port_Open(OB) then
      port.part_of(OB).port_open := FALSE;
      os_files.close(part_of(OB).port_FD);
    end if;
    return port.Get_Port_Open(OB);
 end Close_Port;
--*************************************************************************
-- METHOD: Flash_Queue
-- PURPOSE: Clear the queue
--*************************************************************************
procedure Flush_Queue  (OB : in common_object.object)  is
   result : integer := 0;
   begin
    if ioctl.ioctl(part_of(OB).port_FD,
                   ioctl.TCFLSH,result'address) = -1  then
        text_io.put_line("fail to flush the buffer");
    end if;
  end Flush_Queue;
```

```
--*****************************************************************************
-- METHOD: Delete_Unmanaged_RS232_Port
-- PURPOSE: Delete object
--*****************************************************************************
procedure Delete_Unmanaged_RS232_Port (OB : in common_object.object)  is
  close_status : boolean;
  begin
    close_status := Close_Port(OB);
  end Delete_Unmanaged_RS232_Port;
--*****************************************************************************
-- METHOD: Read_From_Port
-- PURPOSE:
--*****************************************************************************
procedure Read_From_Port(OB                : in      common_object.object;
                         Buffer            : in out  string;
                         num_chars_to_read : in      integer;
                         count             : out     integer) is
   T   : string(1..num_chars_to_read + 1);
   len : integer := T'last;
   begin
     T(1..len-1) := Buffer;
     T(len)      := ascii.nul;
     count := unix.read(part_of(OB).port_FD,
                        T'address, num_chars_to_read);
     buffer := T(1..len - 1);
    end Read_From_Port;
--*****************************************************************************
-- METHOD: procedure Write_To_Port
-- PURPOSE:
--*****************************************************************************
procedure Write_To_Port(OB                 : in     common_object.object;
                        buffer             : in out character;
                        num_chars_to_write : in     integer) is
  begin
    os_files.write(part_of(OB).port_FD,
                buffer'address, num_chars_to_write);
  end Write_To_Port;
end Unmanaged_RS232_Port;
```

Bibliography

1. *Ada 9X Project Report (DRAFT): Ada 9X Mapping Document Volume 1 Mapping Rationale,* MA: Intermetrics, Inc. 1992.
2. Ann L. Winblad et al., *Object-Oriented Software.* New York: Addison-Wesley, 1990.
3. *AT&T C++ Language System Release 2.0 Product Reference Manual,* AT&T, 1989.
4. *AT&T C++ Language System Release 2.0 Selected Reading,* AT&T, 1989.
5. Atkinson, Colin. *Object-Oriented Reuse, Concurrency and Distribution: An Ada Based Approach.* New York, NY: ACM Press, 1991.
6. *Ada and C++ Business Case Analysis.* Deputy Assistant Secretary of the Air Force (Communications, Computers, and Logistics) Washington D.C 20330-1000, July 1991
7. Booch, Grady. *Object-Oriented Design With Applications.* Redwood City CA: The Benjamin/Cummings Publishing Company, Inc., 1991.
8. Booch, Grady. *Software Engineering with Ada (Second Edition).* Redwood City CA: The Benjamin/Cummings Publishing Company, Inc., 1986.
9. Cardelli, L. and P. Wegner *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, 17:471-522 (December 1985).
10. Coad, Peter and Edward Yourdon. *Object-Oriented Analysis (Second Edition).* Englewood Cliffs NJ: Yourdon Press, 1991.
11. Cohen, Norman H. *Ada as a Second Language.* New York: McGraw-Hill, 1986.
12. *CONVEX Interlanguage Programming Guide.* Richardson TX: CONVEX Press, 1992.
13. Cox, B. *Object-oriented Programming: An Evolutionary Approach.* New York: Addison-Wesley, 1986.
14. *Developing And Using Ada Parts Real-Time Embedded Applications.* McDonnel Douglas Missile Systems Company, 1990.
15. Filer, Capt Robert E. *A 3-D Virtual Environment Display System. MS Thesis.* AFIT/GCS/ENG/89D-2. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.
16. Goodenough, J. B. Ross, D. T. and C. A. Irvine *Software Engineering: Process, Principles, and Goals.* IEEE Computer, 8:17-27 (May 1975).
17. Henderson-Sellers, B. and J.M. Edwards *The object-oriented systems life cycle.* Communications of the ACM, 33:142-149 (September 1990).
18. Hooper, James W. and Rowena O. Chester *Software Reuse: Guidelines and Methods.* New York, NY. Plenum Press, 1991.
19. Horowitz, Ellis and J. B. Munson. *An Expansive View of Reusable Software.* IEEE Transactions on Software Engineering, SE-10: 488-493 (September 1984).

20. Khoshafian, Setrag and Razmik Abnous. *Object Orientation: Concepts, Languages, User Interfaces.* New York: John Wiley & Sons, Inc. 1990.

21. Korson, Tim and John D. McGregor. *Understanding object-oriented: a unifying paradigm.* Communications of the ACM, 33:41-60 (September 1990).

22. Krueger, Charles W. *Software Reuse.* ACM Computing Surveys, 24:131-183 (June 1992).

23. Lippman, Stanley B. *C++ Primer (Second Edition).* New York: Addison-Wesley, 1991.

24. Meyer, B. *Object-Oriented Software Construction.* London, U.K.: Prentice-Hall International, 1988.

25. Meyer, B. *Reusability: The case for Object-Oriented Design.* IEEE Software, 4:50-64 (March 1987).

26. Olson, Capt Robert A. *Techniques To Enhance the Visual Realism of a Synthetic Environment Flight Simulator. MS Thesis.* AFIT/GCS/ENG/91D-16. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

27. Phillips, Dwayne. *OOP: now its hot. (object-oriented programming) (Tech Section) (Tutorial).* Computer Shopper, 11:734-735 (Nov 1991).

28. Piper, Joanne *DoD Software Reuse Vision and Strategy.* CrossTalk: The Journal of Defense Software Engineering, 37:2-8 (October 1992).

29. Pressman, Roger S. *Software Engineering, A Practitioner's Approach.* New York: McGraw-Hill Book Company, 1987.

30. Richard Wiener and Richard Sincovec. *Software Engineering with Modula-2 and Ada.* New York: John Wiley & Sons, Inc. 1984.

31. Riehle, Richard. *Object lessons: what we mean when we talk about software engineering. (Objectively Speaking).* HP Professional, 5:76-78 (November 1991).

32. Rumbaugh, James and others. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice Hall, 1991.

33. Schonberg, Edmond *Contrasts: Ada 9X and C++.* CrossTalk: The Journal of Defense Software Engineering, 36:12-16 (September 1992).

34. Scott, McCoy L. *Binding and Ada.* Ada Letters, 8:156-160 (November/December 1990).

35. Shlaer, Sally and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World Data.* NJ: Yourdon Press, 1988.

36. Silicon Graphics, Incorporated. *Graphics Library Programming Guide.* version 4.0 Mountain View, CA, 1990.

37. Simpson, Dennis Joseph. *An Application of the Object-Oriented Paradigm to a Flight Simulator. MS Thesis.* AFIT/GCS/ENG/91D-22. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

38. Stroustrup, B. *What is Object-Oriented Programming?* IEEE Software, 10-20 (May 1988).

39. Taft, Tucker S. *Multiple Inheritance in Ada 9X* CrossTalk: The Journal of Defense Software Engineering, 38:11-13 (November 1992).

40. Taylor, David. *Managing the Transition to Object-Oriented Technology* ACM SIGPLAN Notices: OOPSLA 1991 Proceedings 26:357-358 (Nonember 1991).

41. *Verdix Ada Development System.* Silicon Graphics Computer Systems version 6.1.0, 1991.

42. Wallace, Robert H. *Practitioner's Guide to Ada.* New York: McGraw-Hill Book Company, 1986.

Vita

Captain Samkyu-Lim was born on November 23, 1962, in Junnam, Korea. He graduated from Marianist High School in Mokpo, in 1982. He entered the Air Force Academy in Seoul, in 1982, where he received Bachelor of Science degree in Aeronautical Engineering. Upon graduation he was assigned as a second lieutenant of the Air Force. In 1986, he completed the Elementary Computer Course which was offered by Education Command for the officers who are assigned as computer engineers. Also, in 1987, he completed the Software Development Education Course which was offered by Korean Institute of Defense Analysis for the computer engineers of government. In 1988, he was assigned to the Headquarter of the Air Force where he served as a software engineer. He entered the School of Engineering, Air Force Institute of Technology of United States, in June, 1991.

Permanent address: 536 Bongho-Ri Dopo-Myen
Youngam-Gun Junnam
South Korea

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE March 1993	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE Toward Reusable Graphics Components in Ada		5. FUNDING NUMBERS
6. AUTHOR(S) Samkyu Lim		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583		8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/93M-03
9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES) Software Technology for Adaptable Reliable Systems (STARS) Suite 400 801 North Randolph Street Arlington, VA 22203		10. SPONSORING MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION AVAILABILITY STATEMENT Distribution Unlimited		12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This thesis demonstrates and illustrates a way of developing reusable graphics software components in Ada associated with a C++/C library. The work was carried out using object-oriented software development techniques that were used to analyze, design and implement a partial flight simulator. The objective of this thesis was to present a way of building reusable software components with Ada in a graphics application environment.
An object-oriented approach was taken in the development of a set of reusable graphics software components for a flight simulator domain. A selection of a set of reusable software components came from domain analysis. These components were analyzed in detail, then redesigned to demonstrate and illustrate the thesis objective. Examples from design and implementation demonstrate how Ada 83 was applied in building reusable graphics software components associated with C++ routines, the limitations of Ada 83, and how Ada9X addresses these limitations.

14. SUBJECT TERMS Interface, Ada, C++, Reusability,Graphics			15. NUMBER OF PAGES 117
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL